QUALITY IMPROVEMENT AND VALIDATION TECHNIQUES ON SOFTWARE SPECIFICATION AND DESIGN

LIU SHUANG

(B.Eng., Renmin University of China, 2010)

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

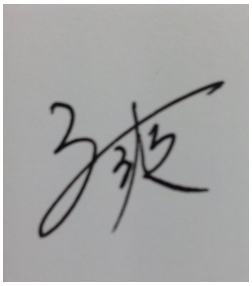
NATIONAL UNIVERSITY OF SINGAPORE

2015

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.



LIU SHUANG

23 March 2015

Acknowledgements

I would like to take this opportunity to express my sincere gratitude to those who assisted me, in one way or another, with my Ph.D study in the past five years.

First of all, I would like to give my most sincere tribute and gratitude to my supervisors Dr. Bimlesh Wadhwa and Dr. Jin Song Dong, for their guidance, encouragement and insights, which guide me through my PhD life; and for their careful reading and constructive criticisms and suggestions on drafts of this thesis. I will always be grateful for their timely help and support during my hard days.

Furthermore, I would like to thank my mentors: Dr. Sun Jun and Dr. Liu Yang. Their academic vision and timely discussions always inspire me. I own special thanks to Dr. Sun Jun, for all the insightful guidance and inspiring discussions.

In addition, I would like to acknowledge the support of my thesis advisory committees: Dr. Siau-Cheng Khoo and Dr. Wei Ngan Chin for their constructive comments on my research. I would like to thank the numerous anonymous referees who have reviewed parts of this thesis prior to publication in conference proceedings.

I would also like to thank all my lab mates in Programming Language and Software Engineer Lab 1. Their help and friendship enriched my life in Singapore.

Last but not the least, I'd like to thank my parents Liu Zunli and Sha Guizhen, for all their love and belief in me.

Contents

Summary

Requirements specification and system design models are the fundamental documents in the software development life cycle. They are the major references for understanding user requirements and to guide later system development and maintenance activities. It has been reported that more than 60% of the errors in software products are introduced during the design phase. Errors introduced in the early phases are much harder and more expensive to detect than errors introduced in the coding phase. It is thus highly desirable to improve the quality of software requirements specifications and design models by detecting software defects as early as possible.

In this thesis, we are motivated to provide techniques to improve the quality of software requirements specifications and design models. For software requirements specifications, we propose two works that focus on improving the quality of use cases, which are widely adopted by different software development methodologies to capture user requirements.

First, we propose to find defects in use case descriptions to improve the consistency and integrity aspects of a single use case. We adopt advanced natural language processing techniques to automatically extract action tuples and predicates from use case sentences. We formally define common defects, e.g., inconsistency and incompleteness related defects, in use case documents and propose algorithms to find those defects based on the automatically extracted action tuples, predicates and the control flow related information. The found defects are linked to the original descriptions in use cases to aid improving the quality of the use case document.

Second, we propose to further improve the use case quality by finding missing scenarios and preconditions/postconditions which involve multiple use cases. We adopt the active learning techniques to learn a Deterministic Finite State Automaton (DFA) for each actor/agent in a use case document. During the learning process, our method finds missing scenarios and missing preconditions/postconditions through interactions with users. The missing scenario is presented as a sequence of actions which is easy to be added to the use case document to improve the integrity of the document.

To find sophisticated, nontrivial errors which may be introduced in the system design phase, we propose to improve the quality of UML state machines models, which are widely adopted to capture the dynamic behaviors of system designs. Our work focuses on finding safety and liveness related defects in UML state machines automatically. We provide an operational semantics for the complete syntax of UML state machines and implement the semantics into the PAT framework, which enables model checking on UML state machines to find liveness and safety related defects.

We evaluated all of our methods with real world documents or models. The evaluation results show that our methods are effective in improving the quality of requirements specifications and design models.

Keywords: **Use Case, Natural Language Processing, Model Checking, Active Learning, UML state machines**

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation and Goals

Software development, one of the key activities in Software Development Life-cycle (SDLC), includes activities such as defining functional requirements, design, coding and testing. Among these activities, capturing functional requirements and system design are the major activities before the real coding phase. They are important for three reasons. Firstly, they serve as the main activities to communicate with stakeholders to understand their requirements. Secondly, they serve as the basis for the later system development phases, e.g., coding, testing and verification. Last but not the least, they also serve as the key reference in the process of maintenance and upgrade after software deployment. It is thus highly desirable to maintain a good quality of the software requirement specifications and design models.

The importance of finding defects[1] in an early development stage and improving the quality

[1]We use the word defect to represent various problems, including inconsistency, incomplete description, deadlock situation, etc., that may be introduced during requirement analysis and system design phase.

of requirement specification and system design models has been well recognized during the past decade. It has been reported that "More than 60% of the errors in a software product are committed during the design and less than 40% during coding." [86], "Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase" [31]. Therefore either for financial savings or system robustness concerns, finding defects in an early stage is of great importance. Actually, successful IT projects have spent about 28% of the effort on the requirement phase [66], which reflects the importance of requirement analysis.

Jacobson et al. [68] proposed a use case driven approach to capture user requirements. Having been pragmatically evolved based on more than 20 years of practices, use case 2.0 [69] now seeks the benefits of "agile, iterative, incremental development at an enterprise level". Use cases have been adopted widely by various different software development methodologies, e.g., Model-driven Engineering and Object-Oriented Software Engineering (OOSE). Use cases are also adopted by the Object Management Group (OMG) and have become one of the UML [7] notations. The major part of a use case document is written in natural language. UML use case diagram, UML activity diagram, UML sequence diagram and UML state machine diagrams are (optionally) used as complements to visualize use cases.

Natural language is imprecise and ambiguous in nature, therefore, defects are inevitably introduced into use case documents. These defects, including inconsistencies and incomplete statements in each use case, may introduce barriers to understanding, which may further lead to ambiguities in model design, failures of software development as well as maintenance problems. Finding defects in use case documents is thus an important task. Traditionally, defects in use cases are inspected manually, which is tedious and error-prone. Moreover, manual inspection cannot meet the increasing demand on the short delivery time. Therefore automatic defect detection in use cases is gaining increasing attention. Recently, several works [55, 56, 118, 117, 134] are proposed to find consistency defects in use cases. The

completeness related issues, e.g., whether alternative flows/conditions and exceptions are addressed thoroughly and clearly, are not considered. Moreover, existing approaches [118, 117, 134] apply document-specific templates on the results of the simple natural language parsing technique, i.e., Part-of-Speech (POS) parsing. Since the templates are document dependent, the application of those approaches are limited.

In addition to the incompleteness which exists within a single use case, there are also incompleteness related problems involving multiple use cases. Since use case is a scenario based technique to capture user requirements, it is always the case that only a partial behavior of an actor/agent is properly described. Missing of scenarios may hinder the understanding of the requirements and hide potential consistency related defects. There are existing works [41, 97, 125, 132] which generate state based transition systems from scenarios captured by Message Scequence Chart (MSC) [4]. However, MSC is a formal structure and is not easy to obtain at the first hand. Usually strong knowledge and experience on UML modeling are required to construct MSC from raw natural language descriptions, which are the initial form of scenarios. Moreover, it is hard for stakeholders to get involved with such a formal structure, which further raises difficulties for specification validation. Another drawback of these approaches is that they all assume the scenarios, which need to be synthesized, would start with the same preconditions. However this is usually not true. In particular, preconditions and postconditions, which capture the valid starting and ending status of a use case, should be properly considered.

In the design phase, various models are usually developed as an abstraction to reflect different aspects of a system. UML state machines are widely used to model the dynamic behaviors of a system. Safety and liveness properties need to be verified on those models in order to uncover design defects. Model checking [38], an automatic verification technique, has shown its potential in automating the formal verification process on both hardware and software designs, especially on verifying the system dynamic behaviors. There are approach-

es which provide model checking support for UML state machines. Those approaches either are based on a formal operational semantics [51, 85, 128, 45] for UML state machines or provide translation rules [22, 26, 33, 36, 84, 104, 139] from UML state machines to existing formal languages such as Abstract State Machine (ASM) [48], Petri Nets [71] and specification languages, such as Promela [13], CSP [3] and CSP# [120], of model checking tools. The operational semantics in existing approaches only cover a subset of UML state machine features. The translation-based approaches depend on the target language as well as the tool support for the target language, thus are fragile to changes on target languages.

Motivated by the importance of improving the quality of requirement specification and design models and the weaknesses of existing works, we are devoted to proposing methods to improve the state of the art. Since we are focusing on a development phase where stakeholders are expected to be actively involved, our methods take active consideration on getting stakeholders to be involved and thus better improve the quality of the requirements and designs.

1.2 Outline and Overview

The main contribution of our work is to propose methods to uncover defects introduced in requirement and design phases early. Our methods reflect the found defects in formats that are easily understandable by stakeholders, thus can directly help to validate and improve the quality of requirement specification and design models.

The remaining of this thesis is organized as follows:

Chapter 2 provides the background knowledge, including basis on software development process, use case and UML state machines, of this thesis.

In Chapter 3, we present our work on early intra-defects [2] detection in use case documents. We explore advanced natural language processing techniques [140] to parse the sentences in the use case descriptions. We then provide analysis rules to analyze the parsing results and automatically extract entities from parse trees. The analysis rules we proposed are based on the general English grammar and have good adaptability compared to document-specific templates. We formally define common defects, considering both consistency and completeness issues, in use case documents. Our methods successfully find defects in the use case documents of a real system and provide defect reports which link the defects to the original use case specification document. The found defects are confirmed by the developers to be real defects.

In Chapter 4, we present our work on improving the quality of use case documents through learning and user interaction. We adopt advanced natural language parsing techniques [140] and active learning techniques [23] to incrementally learn a DFA from the behaviors in use case scenarios. Our methods find potential missing scenarios, preconditions and postconditions during the process of active learning, through interactions with users. The interaction with users is presented in the format of action sequences in natural language, which greatly improves the involvement of users.

In Chapter 5, we present our work on model checking aided design validation. To be specific, our focus is on UML state machines. We propose an operational semantics for the complete syntax set of UML 2.4.1 [7] state machines. Our proposed semantics cover all the syntax features of the latest UML state machine specifications and respect to the UML state machine metamodel. We implement the semantics in a self-contained model checker USMMC [92], which enables model checking on UML state machines. We compare our tool with an existing UML state machine model checking tool HUGO [18] and the results show that our tool outperforms HUGO on all the UML state machine models we adopted from

[2] defect within a single use case

the literature.

In Chapter 6, we review the existing approaches that are related to this thesis. We discuss the differences between our work and those related work and summarize our improvements on state-of-the-art techniques.

We conclude in Chapter 7. Future research directions are also discussed in this chapter.

1.3 Acknowledgment of Published Work

Most of the work in this thesis are published in international conference proceedings or submitted for review.

- **Automatic Early Defects Detection in Use Case Documents** [93] is published in Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14). This work is presented in Chapter 3.
- **A Formal Semantics for Complete UML State Machines with Communications** [91] is published in The 10th International Conference on integrated Formal Methods (iFM'13). This work is presented in Chapter 5.
- **USMMC: A Self-Contained Model Checker for UML State Machines** [92] is published in The 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13). This work is presented in Chapter 5.

Moreover, the work related to improving use case documents through leaning and user interaction, which is presented in Chapter 4, is submitted to a peer-reviewed conference for review.

Chapter 2

Background

In this section, we briefly introduce the general background knowledge that is referred to in this thesis.

2.1 Software Development Process

Software development, one of the key activities in Software Development Life-cycle (SDLC), includes activities such as defining functional requirements, creating high level/module design, coding and testing. Among these activities, capturing functional requirements and system design are the main activities which help to understand users' requirements and link user requirements with coding and subsequent development steps.

Due to the variety of software products, different software development models, such as waterfall model [29], spiral model [32] and agile model [21] are proposed to fulfill software development process. It is up to the software development teams to choose a proper model for their development. Although developers may choose different development models according to their expertise or company convention, some development activities, such as

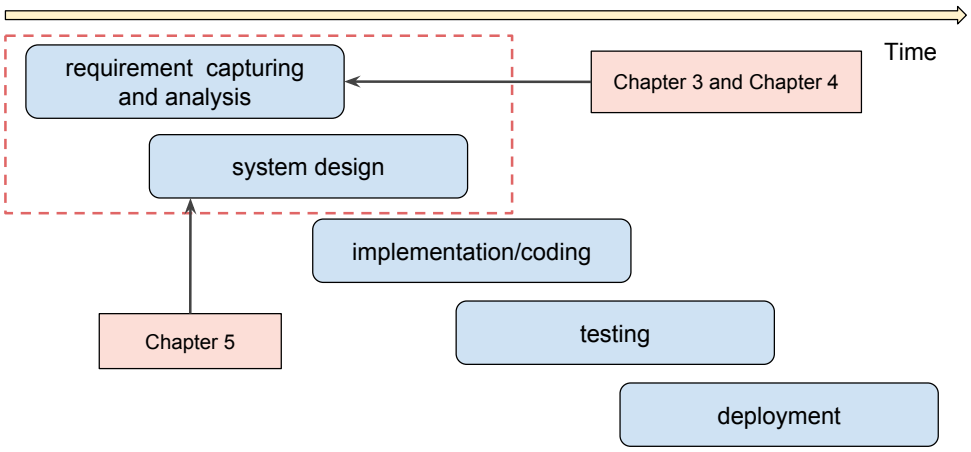


Figure 2.1: Common activities in software development

those shown in Figure 2.1, are commonly adopted in the software development process. In Figure 2.1, the horizontal-axis represents time and the rectangles represents software development activities. We do not use any arrows to link those activities since in different software development models, different iterations and overlapping of those activities may happen. However, the general ordering of activities follows what is shown in Figure 2.1.

In this thesis, we focused on the first two activities, i.e., requirement analysis and system design, in the software development process. Chapter 3 and Chapter 4 discuss our work on improving the quality of requirement specifications captured by use cases. Chapter 5 discusses our work on improving the quality of design models, specifically dynamic behavior models captured by UML state machines.

2.2 Use Case

Use case, since proposed by Jacobson [68], has been adopted by many software development methodologies. Use case is not only a technique to capture requirements, it is like the hub of a wheel [67] which binds together many software development activities, including

Use Case 1: Receive the order with special group
Initiating Actor: Trader
Pre-Conditions
1. The order is legal.
Main Flow
41. GSYS receive the symbol of order.
42. Check the order.
43. If the order is legal, record values of the group.
44. Find the constraint in the system according to the group name.
45. Save the order into database.
46. Price the order.
47. During the processing, it could create matches only when the constraints are permitted. For example, no match should be created if there is not enough cash in the group.
48. This ends the use case.
Alternative Flow
In step 4, if there is no such constraint in the system, the system will reject this order.
Post-Conditions
1. Order with special group is received by the system.

Figure 2.2: Example of use case description

requirements, analysis and design, testing, etc. A use case typically contains a list of steps which define the interactions between an actor and a system. The major part of a use case is described in natural language.

An example natural language use case description is shown in Figure 2.2. There are six major sections, including use case name, actor/agent, precondition, main action flows, alternative action flows and postcondition, in a use case description. The main flow section captures the normal execution flows. The alternative flow section captures alternative execution flows when certain conditions in the main flow are not satisfied.

There is no standard template for writing use case documents as concluded by Fowler [54]. The choice of use case styles may be highly project-dependent as affected by factors such as the criticality and the number of people involved in the project. However, there are guidelines [39] in choosing different styles of use cases for different projects. It is recommended that for small projects (4-6 people involved), a simple, casual use case template [39] can

be chosen. For large, life-critical projects, it is more appropriate to use a hardened, fancier and fully-addressed template [39]. The casual use case template has a high tolerance in writing styles and structures, thus is usually verified manually. In contrast to the casual use case template, a fully-addressed use case template is less tolerant and requires people to adhere to the template (structure, grammar, naming conventions, etc.) closely. Since a fully-addressed use case template is usually adopted by large projects, which are often life-critical, we focus on fully-addressed use cases in this thesis.

There is no strict, universally adopted fully-addressed use case templates. Various writing styles [39] (e.g., Cockburn, RUP, one column table, 2-column table, If-statement style, etc.) have been proposed. However, it has been reported by Cockburn [39] that "the readers almost universally select the single-column, numbered, plain text, full sentence form". Therefore, in this thesis we focus on this most popular writing style in literature. We checked the use case template used in industry [2] and found that those templates are consistent with the template in [39] in majority of the sections which capture functional requirements. Figure 2.2 is one use case for a stock trading system[1] which follows roughly the Cockburn style [39].

In addition to the natural language descriptions, UML diagrams, such as use case diagram, activity diagram and sequence diagrams may be used to visualize a use case. For example, use case diagrams provide a high-level view, which capture the interactions between actors and the system as well as the relations (extension/inclusion/generalization) between use cases. Activity diagrams are usually used to visualize the (conditioned) stepwise actions of a use case. Sequence diagrams are usually used to describe the interactions between different actors in one or multiple use cases.

[1] We omit the name of the system due to the confidentiality. The use case is modified to hide the sensitive keywords.

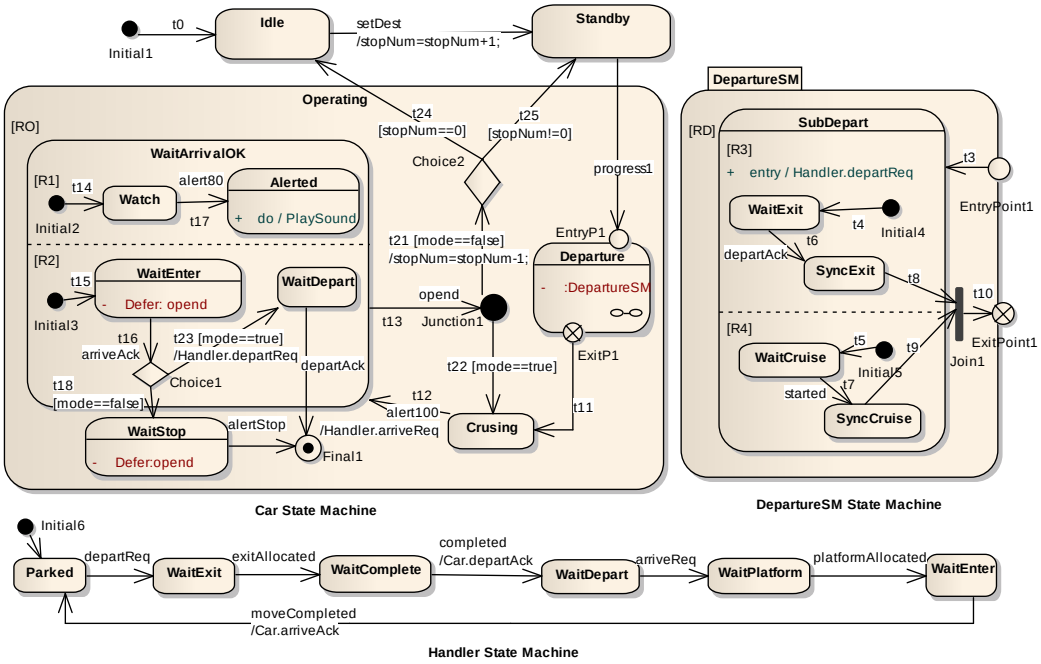


Figure 2.3: The RailCar state machine

2.3 UML State Machines

UML state machines [7], a variance of Harel statechart [61], are widely used to capture the dynamic behaviors of a system design. Figure 2.3 shows a UML state machine for the *RailCar* system (a modified version of the example used in [62]). The *RailCar* system is composed of 3 state machines: *Car*, *Handler* and *DepartureSM* (referenced by the *Departure* submachine state in the *Car* state machine). The *Handler* state machine models a part of the terminal behavior, which is responsible for communicating with the *Car* state machine when the car is approaching and departing the terminal. They communicate with each other through synchronous event calls. We use the *RailCar* state machine as a running example to illustrate the basic features of UML state machines.

UML state machines have three kinds of features/constructs, i.e., vertex, regions and transitions.

Vertex. UML state machine uses the concept vertex to represent all nodes in the graphical notation. Therefore a vertex is the general designation of state, pseudostate, final state and connection point reference which are introduced below.

Transitions. A Transition is a relation between a source vertex and a target vertex. In Figure 2.3, the arrow labeled t_0 is a transition. Guards, triggers and effects are associations of a transition. A guard (e.g., *mode==true* of transition t_{23} in Figure 2.3) is a boolean constraint which must be evaluated to true in order to fire the corresponding transition. A trigger (e.g., *opend* of transition t_{13} in Figure 2.3) relates an event to a behavior and will cause execution of the behavior when the event specified by the trigger occurs. An effect (e.g., $stopNum = stopNum - 1$ of transition t_{21} in Figure 2.3) is a behavior, which is a sequence of actions [2]. The container of a transition is the region which owns the transition. A compound transition is composed of a multiple transitions joined via choice, junction, fork and join pseudostates.

Regions. It is a container of vertices and transitions, and represents the orthogonal parts of a composite state or a state machine. In Figure 2.3, the area *[R1]* is a region.

States. There are three kinds of states, viz., simple state (e.g., state *Idle* in Figure 2.3), composite state (e.g., state *Operating* in Figure 2.3) and submachine state (e.g., state *Departure* in Figure 2.3). An orthogonal composite state (e.g., state *WaitArrivalOK* in Figure 2.3) has more than one region. States can have optional entry/exit/do behaviors. A do behavior (*PlaySound* in state *Alerted*) can be interrupted by an event. A state can also define a set of deferred events ({*opend*} in state *WaitEnter*). A final state (*Final1* in Figure 2.3) is a special kind of state which indicates finishing of its enclosing region.

Pseudostates. Pseudostates, e.g., initial, join, fork, junction, choice, are introduced to

[2] Action is a basic unit of behavior specification. Actions include send/receive messages, update values and so on.

connect multiple transitions to form complex transition paths. An initial pseudostate (*Initial1* in Figure 2.3) is used to indicate the default active vertex for each region of a composite state, it cannot act as the target of a transition. A join pseudostate (*join1* in Figure 2.3) is used to merge transitions from states in orthogonal regions. A fork pseudostate is used to split transitions targeting states in orthogonal regions. A Junction pseudostate is introduced as syntactic sugar to merge/split incoming transitions into outgoing transitions. It represents a static branching point. A Choice pseudostate (*Choice1* in Figure 2.3) represents dynamic branching points, i.e., the evaluation of enabled transitions is based on the environment when the choice pseudostate is reached.

Connection Point Reference. It is an entry/exit point of a submachine state and refers to the entry/exit pseudostate of the state machine that the submachine state refers to. In Figure 2.3, *EntryP1* and *ExitP1* in *Departure* state are connection point references.

Active State Configuration. An active State configuration is a set of active states of a state machine when it is in a stable status[3]. In Figure 2.3, {*Operating, Crusing*} is an active state configurations.

Run to Completion Step (RTC). It captures the semantics of processing one event occurrence, i.e., executing a set of compound transitions (fired by the event), which may cause the state machine to move to the next active state configuration, accompanied by behavior executions. It is the basic semantic step in UML state machines. For example in Figure 2.3, {*Operating, WaitArrivalOK, Watch, WaitDepart,*} $\xrightarrow{opend}$ {*Idle*} is an RTC step.

[3]The state machine is waiting for event occurrences.

Chapter 3

Finding Intra-defects in Use Case Descriptions

Use cases [67] are the main technique for understanding user requirements, which have been widely adopted in the modern software development life cycle over the last two decades. Driven by the necessity of communicating with stakeholders, the majority of a use case document is written in natural language, which inevitably introduces defects. In this chapter, we discuss our method on finding intra-defects [1] in natural language use case descriptions.

3.1 Introduction

A use case describes a sequence of interactions between a software system and an external actor such that the actor is able to achieve some goal. Collectively, use cases are used to define all the necessary system activities that have significance to the users. As use cases are developed during a very early stage of the software development life cycle, they also serve as

[1] Defects present within a single use case description.

the basis for developing detailed functional requirements, help in design development and validation, system testing, maintenance of evolving of the software, and even in creating an outline for user manuals. High quality use case documents can improve the sustainability of software.

Use case documents are usually written in natural languages, which may inevitably introduce defects like inconsistency, redundancy and incompleteness. Moreover, those defects are hard to identify or verify due to their informal format. Software engineers actually enjoy the flexibility of natural language descriptions on use cases (compared to formal descriptions) since they can communicate more smoothly with stakeholders in this way. On the other hand, natural language descriptions of use cases make it challenging to analyze and validate the requirements, which are necessary in a mature requirement engineering methodology. In the current practice, use case analysis is conducted manually, e.g., requirement analysts manually extract analysis models (e.g., state machine, activity diagram and sequence diagram) from the use cases, and then search for defects in the models or validate them against test cases. Manual analysis is hardly ideal as it requires a lot of human efforts and is often error-prone. As a result, use cases are much less useful than they could or should be.

There are existing works on automatic analysis of use cases [56, 77, 78, 118, 137]. But still, we identify the following challenges which have not been addressed satisfactorily.

Firstly, actual use case documents are often larger and more complex than those have been reported in existing works [77, 78, 137]. For large use case documents, the diversity of grammar rules and ambiguities presented in the document raise great technical challenges in automatically "understanding" them. For example, one of the documents that we used for our evaluation[2] contains 188 use cases and more than 1700 sentences. The diversity of sentence styles as well as the grammar errors in the sentences make it challenging

[2]This is a real system used for real-time stock trading in the amount of billions. The document is provided by our industry collaborator.

to provide templates for parsing. Some existing approaches rely on heuristics or human intervention [77, 78], which may not be feasible for large use case documents.

Secondly, common problems in use cases are inconsistency and incomplete flows. Existing approaches have so far mainly focused on analyzing inconsistency problems [56, 118] and leave incomplete flows unconsidered. A further issue is that there have been limited formal definitions on what is regarded as defects/errors.

Lastly, some existing approaches (e.g., [56, 134]) rely on users to provide use-case-specific templates for parsing, which is ad-hoc and may require knowledge about shallow parsing techniques. The most difficult challenge is to develop a method (and perhaps a tool) which achieves good accuracy in understanding use cases and detecting problems, and at the same time, is able to be generalized to work with use cases in different domains.

In this chapter, we are motivated to provide automatic techniques to identify defects, i.e., inconsistency and integrity related problems, in a natural language use case description. We contribute in the following three aspects.

- We explore dependency parsing technique to help understand use case documents. We provide 8 rules based on general English grammar to analyze the dependency parsing results.
- We formally define common inconsistency and integrity related defects in use cases and provide algorithms to automatically check those defects. Horizontal tractability links to the original use case document are preserved for user consumption.
- We conduct experiments with use case documents of 5 different systems from different application domains. The results show that our method can achieve good accuracy in analyzing sentences from different domains as well as in finding defects.

Outline. Section 3.2 provides preliminaries used in this chapter. We briefly walk through

our approach, with an example, in Section 3.3. The technical details of our approach are then discussed in Section 3.4. The experimental results are reported in Section 3.5. Section 3.6 discusses the limitations, manual efforts as well as theads to validity in the evaluation. Section 3.7 provides conclusions.

3.2 Preliminary

This section introduces the preliminaries of definitions in use cases and the UML activity diagram used in this thesis.

3.2.1 Definitions in Use Cases

As discussed in Section 2.2, there are a variety kind of use case templates. Our work do not aim at handling all the possible writing styles of use cases. We are rather interested in investigating advanced NLP techniques to aid defects detection in use case documents. In this thesis, we are devoted to processing use cases following the fully-addressed use case template (e.g., Figure 2.2), which is the most popular adopted template in practice. The issues caused by different writing styles can be tackled by providing more robust pre-processing steps. Our work focuses on the core sections, including use case name, actors, preconditions, main flow, postconditions and alternative flow, in use case documents (following a fully-addressed use case template). We formally define the concepts involved in use case descriptions below.

Definition 1 (Action) *The action is defined as* $A \triangleq (vb, sub, obj)$*, where* vb, sub, obj *are natural language phrases representing the main verb, subject and object of the sentence.*

For example in Figure 2.2, the action tuple of the second sentence in main flow section is $(check, \_, order)$ (The subject is missing in an imperative sentence).

Definition 2 (Predicate) *The* predicate *is defined as* $P \triangleq (ar, R, a_1, a_2)$*, where* $ar \in \{1, 2\}$ *is the arity of the predicate;* R *is the relation symbol of the predicate;* a_1 *and* a_2 *are the arguments of the relation symbol.*

The predicate can be monovalent or divalent, depending on the structure of the sentence. Predicates of higher arity are not used as frequently as monovalentor/divalent predicates. Therefore we do not consider predicates with more than two arities in our work. To gain an intuitive view, a monovalent predicate $(1, is\_legal, order)$[3] can be generated from the sentence in the preconditions section in Figure 2.2. We extract predicates from the preconditions and postconditions sections of the use case description. The guard condition of a sentence is also represented as a predicate.

Definition 3 (Sentence) *A* sentence *is defined as a tuple* $S \triangleq (s\#, \alpha, c, n_s, n_j, UC_{ref})$*, where* $s\#$ *is the sentence number in the corresponding section of the use case;* $\alpha \in A$ *is the action of the sentence;* $c \in P$ *is the guard condition for executing the sentence;* $n_s \in N$ *and* $n_j \in N$ *are the logical previous and succeeding sentence of the current sentence respectively.* UC_{ref} *is the use case name that is referred to by the sentence.*

For example the alternative flow sentence in Figure 2.2 corresponds to the following sentence structure: $(a_1, (reject, system, order), (2, is\_no, there, constraint), 3, -1, NULL)$. a_1 is the step number of the sentence. $(reject, system, order)$ is the action to be conducted in this step. $(2, is\_no, there, constraint)$ is the condition predicate which should be satisfied in order to conduct the action. The number 3 indicates that the current alternative flow step starts from main flow step 3. -1 indicates that there is no explicit assigned step after the current step, then the flow goes to the next neighboring step. There is no use case associated (through include/extend relation) with this use case, therefore the last field is *NULL*.

[3]We use underline to replace spaces.

Definition 4 (Use Case) *A* use case *is defined as a tuple* $UC \triangleq (UCName, PreC, PostC, MF, AF)$*, UCName is the name of the use case;* $PreC \subset P$ *and* $PostC \subset P$ *are the predicates extracted from sentences in the precondition and postcondition sections; MF and AF are the list of sentences S in the main flow and alternative flow sections of the use case.*

3.2.2 UML Activity Diagram

UML Activity Diagrams [7] are used to model the sequence and conditions for the purpose of coordinating low-level behaviors. They are commonly adopted to describe the event flows in use case documents. An action in an activity diagram represents a single activity. They can be expressed in application-dependent languages. In this chapter, we use action tuples (Definition 1) to represent actions.

Definition 5 (Activity Node) *An* activity node *is defined as* $N \triangleq N_a \cup N_c$ *where* $N_a \triangleq (n, \alpha)$ *is action node and* $N_c \triangleq (n, t)$ *is the control node. n is the name for each node.* $\alpha \in A$ *is the action associated with the action node.* $t \in \{decision, final, initial\}$ *is the type of the control node.*

In Figure 3.1, all the rounded rectangles are action nodes. A choice node is represented as a diamond and the enriched circle represents the final node. The solid circle is the initial node. They all belong to the control node set.

Definition 6 (Activity Edge) *An* activity edge *is defined as* $E \triangleq (sn,\ tn,\ g)$*, where* $sn \in N$*,* $tn \in N$ *and* $g \in P$ *are the source, target nodes and the guard condition of the activity edge.*

The guard condition for an activity edge must be satisfied in order to fire the corresponding edge.

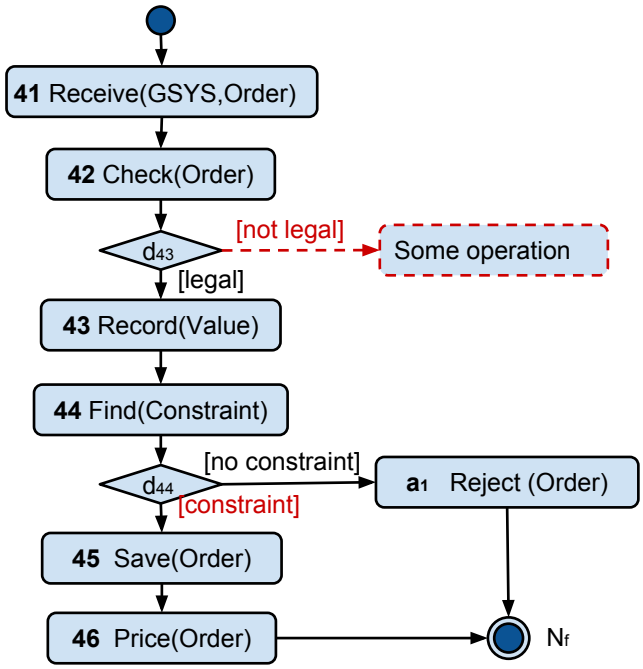


Figure 3.1: Example activity diagram

Definition 7 (Activity Diagram) *A UML activity diagram is defined as* $AD \triangleq (ADName, PreC, PostC, AN, AE)$*, where* $ADName$ *is the name of the activity diagram.* $AN \subset N$ *and* $AE \subset E$ *are the set of activity nodes and activity edges in the diagram.* $PreC \subset P$ *and* $PostC \subset P$ *are the preconditions and postconditions of the activity diagram.*

In this chapter, we consider the subset of UML activity diagram features which are related to control flows as defined in Definition 7. The features which capture object flows, such as object nodes, are not considered since our defects detection methods utilize only the control flow information in the activity diagram.

3.3 Overview of Our Approach

The overview of our approach is illustrated in Figure 3.2. The rectangles represent artifacts that are produced as (intermediate/final) processing results. The ellipses represent the

processing steps. Our method consists of two phases. In the first phase, we take a use case document as input and parse each sentence in the document into parse trees (dependency tree and phrase structure tree). The second phase takes parse trees as input and generates a UML activity diagram for each use case. Afterwards, defects in the use cases are checked. The output of our method includes the UML activity diagrams and a defect report where all defects with horizontal links to the original document are listed. There is also an optional phase as enclosed in the dashed lined area. This phase provides a way to train a domain-adaptive dependency parser in order to improve the accuracy of dependency parsing. In this section, we illustrate our approach with the running example shown in Figure 2.2. We discuss the details in Section 3.4.

Step 1: Pre-processing the document In this step, we remove the irrelevant information and formatting symbols, such as parenthesized comments and bullets, which may affect the parsing accuracy. For example, in Figure 2.2, sentence 41 in the main flow section will be "41\n GSYS accepts the symbol of order .\n" after pre-processing. This is a general process applicable to any document.

Step 2: Free text parsing We use an advanced statistical natural language (dependency and phrase structure) parser ZPar [140] to parse the pre-processed sentences. The dependency parser (Step 2.1) is used to extract bootstrap information for action tuples and the phrase structure parser (Step 2.2) is used to identify the modified and supplement information. The output format of the dependency parser is a dependency tree. Figure 3.3 shows the dependency tree for the first sentence in the main flow section in Figure 2.2. The middle row is the original sentence (in tokenized words). The last row is the Part-Of-Speech (POS) tags of the corresponding words. The labeled links are dependency relations between two words. For example, the link from the word "GSYS" to the word "accepts" labeled with SUB represents that "GSYS" is the subject of "accepts". The word "accepts" is the ROOT, i.e., the main verb of the sentence. In addition to dependency parsing, we also utilize the

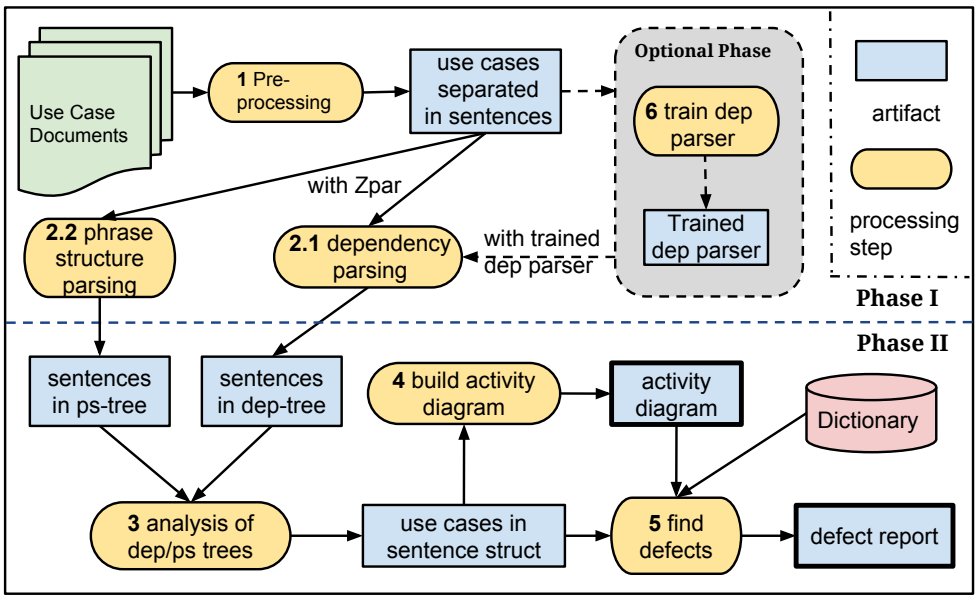


Figure 3.2: Overview of the defect detection approach

phrase-structure parser of ZPar to parse each sentence in the document. The parsing result is a phrase structure tree as shown in Figure 3.4. The leaf nodes are the plain text tokens. The non-leaf nodes are POS tags, where "S", "VP", "NP" represents a sentence, verb phrase and noun phrase respectively. The phrase structure tree is used in combination with the dependency tree in our analysis phase to obtain more accurate results. For example in Figure 3.3, we identify that the object is "symbol" from the dependency tree. The phrase structure tree in Figure 3.4 provides the complementary information that the "symbol" is an attribute of the "order". This kind of attributive information is useful in comparing action tuples.

Step 3: Analyzing parse trees We analyze the dependency trees and the phrase structure trees to extract useful information, including the phrases that capture control flow information, actions and conditions. For each sentence, we extract subject, object, main verb, conditions, and the previous/next step of the current action. For example, for the sentence in the alternative flow in Figure 2.2, 3, $(reject, system, order)$ and $(is\_no, constraint)$

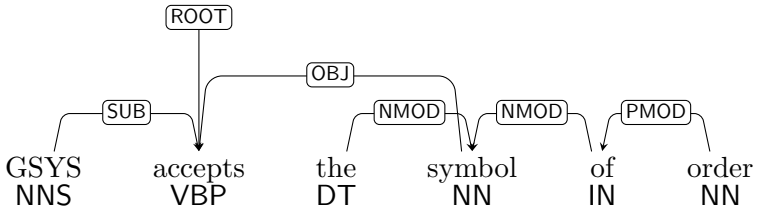


Figure 3.3: Example of a dependency tree

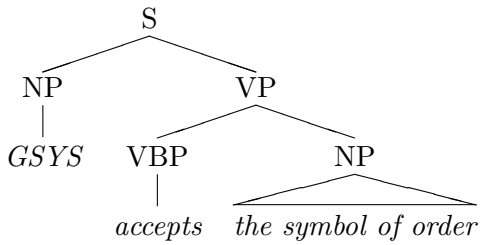


Figure 3.4: Example of a phrase structure tree

are identified as the previous step, the action and the condition information respectively. The subject, object and main verb of an action tuple can be obtained immediately from a dependency tree since they are captured by the dependency relations. For example in Figure 3.3, "symbol" is identified as the object by analyzing the dependency tree directly. However, identifying only this single word looses information, i.e., the attributive adjunct of the word "symbol", which we may need for the later defects detection. Therefore, we query the phrase structure tree of the same sentence (in Figure 3.4) to obtain the noun phrase that the word belongs to. After the adjustment, the action tuple corresponding to the sentence in Figure 3.3 is $(accepts, GSYS, symbol\_of\_order)$. We record such information for each sentence so as to build activity diagrams in the next step.

Step 4: Building activity diagrams We build an activity diagram for each use case based on the identified information in step 3. Figure 3.1 shows the activity diagram that is generated from the use case in Figure 2.2 by our approach. The action node is labeled with the step number and the action tuple extracted from the corresponding sentence. The decision node (diamond) is labeled with the step number of the sentence in which it is

generated. For example the decision node labeled d_{43} is generated from the sentence with step number 43. The guards, edges and nodes in dashed line in figure 3.1 represent the missing flow step that our method detected. The main flow step labeled 47 in Figure 2.2 does not have a corresponding action node in the activity diagram. The reason is that it specifies some constraint and example instead of describing an action step, thus is regarded as irrelevant contents in the step flows and is removed during the activity diagram building procedure.

Step 5: Finding defects We proposed defects detection methods for each defect type defined in Section 3.4.5. Some defects can be found in the use case structure itself. The others, which are related to control flow information are found in the activity diagrams generated by our method. For example, in the use case shown in Figure 2.2, the sentence in the alternative flow refers to "step 3", which is not present anywhere in the use case. This is detected as an inconsistent step numbering defect. As another example, the dashed edge and node in Figure 3.1 show an missing alternative flow step of the use case in Figure 2.2. The output of the defect finder is an error report which contains the error type as well as horizontal links to the original document.

Step 6: Training Dependency Parser To handle the problem caused by document-specific factors, such as grammar errors and specific sentence structures, we provide a way to train a domain-adapted dependency parser. In the case of the stock trading system, we manually labeled a small percent (to be precise, 6%) of wrongly labeled sentences randomly selected from the document to train a domain-adapted dependency parser. This is shown in the dashed box (step 7) in Figure 3.2. The trained dependency parser will replace the ZPar dependency parser in the dependency parsing step. This is an optional step in our overall procedure and is only needed in order to achieve higher accuracy on document specific patterns.

3.4 Approach Details

In this section, we discuss the details of each step in our approach.

3.4.1 Pre-processing Use Case Documents

This step is conducted to filter noises from the input document so as to improve the accuracy of the dependency parser. The output text satisfies the following conditions.

1. Each sentence is stored in a separate line.
2. Each punctuation is preceded by a space.
3. Step index number is stored in a separate line.
4. Parenthesis are replaced by "–LBR–" or "–RBR–".
5. There is no empty line in the document.

We utilized splitta [14] to process (1) and (2), and regular expression matching to perform the other filtering tasks. Some information, such as the section indicator (Pre-Conditions, Main Flow, etc.) shown in Figure 2.2, is use-case-specific. Different development teams may use different notations for the same section. We thus allow use-case-specific-configuration on those indicators so as to flexibly support use cases written by different development teams.

3.4.2 Free Text Parsing

In this chapter, we leverage on ZPar [140], a statistical dependency and phrase structure parser, for analyzing syntactic information, as opposed to using Part-Of-Speech (POS) tags adopted in existing approaches [77, 118, 117]. ZPar utilizes the Wall Street Journal

sections of the Penn Treebank [98] as training data, deriving its disambiguation model using supervised learning.

Dependency Parsing The dependency parser analyzes natural language texts according to the dependency grammar proposed in [106]. It conducts statistical analysis on POS tags based on a large data set, which guarantees that it provides more general results than directly analyzing POS tags based on the templates extracted from the sample document. The dependency parsing technique can also provide richer syntactic details, i.e., the dependency relation between pairs of words, which provide the subject, object and main verb information of a sentence directly and thus reduce the efforts of template matching on POS tags to identify those composition. Moreover, when sentences have a variety of structures, dependency parsing is more robust and more adaptive.

Phrase-structure Parsing Dependency parsing focuses on relations between words. Sometimes a single word cannot provide enough information for defects detection. Phrase structure parsing provides a parse tree in which sentences are parsed into noun/verb phrases and sub-sentences based on the subordinating and modification relations, thus can provide complete context information for an identified word. For each sentence, we analyze the dependency tree to identify bootstrap indicators, i.e., subject and object. Then we use those bootstrap indicators to identify the subordinating noun phrases in the the phrase structure tree. This information is served as context information for our later analysis.

3.4.3 Analyzing Parse Trees

Before discussing the detailed analysis techniques, we formally define a dependency tree first.

Definition 8 (Dependency Tree) *A* dependency tuple *is defined as* $DT \triangleq (T, POS, PI, L)$, *where* T *is the word text;* POS *is the part-of-speech tag for the word;* PI *is the parent index*

of the word and L is the dependency label. A dependency tree *DTree is a list of* dependency tuples.

For the *POS* and *L* fields, we follow the Penn Treebank convention. The dependency tree in Figure 3.3 can be represented as the list ((“GSYS”, NNS, 1, SUB), (“accepts”, VBP, -1, ROOT), (the, DT, 3, NMOD), (“symbol”, NN, 1, OBJ), (“of”, IN, 3, NMOD), (“order”, NN, 4, PMOD)), where indices of the tuples start from 0.

The goal of analyzing a dependency tree is to extract the sentence structure S as defined in Definition 3. The sentence step number ($s\#$), step start (n_s) and join (n_j) nodes are extracted based on keyword matching. We assume that a limited number of indicators, such as “step”, “go to”, “return”, are usually used to indicate control flow changes. We achieve a high accuracy by keyword matching on extracting those fields in all the 5 documents that we used for our evaluation. The α field of a sentence captures the action that the sentence should conduct and the c field is the condition predicate that must be satisfied in order to conduct the action. We discuss the action extraction and predicate extraction method in details.

Extract Action Tuples An action tuple, as defined in Definition 1, contains a subject, an object and a main verb of a sentence. These parts are immediately available in a dependency tree. For example, in Figure 3.3, the dependency labels SUB, OBJ and ROOT indicate that the subject, object and main verb are “GSYS”, “symbol” and “accepts” respectively. However, dependency parsing suffers from a common problem of natural language parsing, i.e., fragile to ambiguities and noises. Thus relying only on the dependency labels may not provide good accuracy. Directly applying ZPar dependency parser resulted in an accuracy of around 44% in identifying action tuples. One reason is that the diversity of sentence patterns, tenses and subordinate structures may cause deviations in the dependency trees. To improve the parsing accuracy, we (1) provide 8 adjusting rules (shown in Table 3.1) based on general

Table 3.1: Rules for extracting action tuples

	Rules
Main Verb	(1) *HAVE* {ROOT}+(*NOT*)+*BE* +(JJ&)+ VB& (2) *BE* {ROOT}+(*NOT*)+ (JJ&) + VB& (3) *BE* {ROOT} +(*NOT*)+ JJ& {PRD} (4) *MODAL* {ROOT}+(*NOT*)+*BE*+(JJ&)+ VB& (5) ROOT {PI=-1}
Subject	(6) *PI.L* = ROOT{SUB}
Object	(7) *PI.L* = ROOT{OBJ/PRD/PRP} (8) *PI.L* = ROOT{SBAR/ VC/ VMOD}

English grammar. (2) rely on the phrase structure parsing result to identify related context information. The rules, shown in Table 3.1, are general in the sense that they are based on natural language grammars and do not contain any document-specific information, e.g., patterns and key words.

There are 3 kinds of information, i.e., plain text, POS tags and dependency label, used in our rules. The plus symbol + composes constraints for consecutive words. We use braces {} to represent compulsory information when more than one kind of information is used on one word. Brackets () are used to represent optional information and the slash / symbol is used to represent a choice among the candidates. For example in Table 3.1, rule (7) requires that the parent of the word should be labeled as ROOT and the word itself should be labeled as OBJ or PRD or PRP. The symbol & represents a group of POS tags, for example, VB& represents the POS tags VB, VBD, VBG, VBN, VBP, VBZ, which are different formats of verbs. The words *HAVE*, *BE* and *NO* in capital italic are plain word tokens, which represent the set of all possible formats of a word. For example, *HAVE* represents the set of words: "have, has, had". *PI.L* =ROOT represents that parent of this node should be labeled as the ROOT. We claim that these rules are general, i.e., not document specific. For example, rules used to identify the main verb of a sentence (*MainVerb*) capture sentence styles "have (not) been done", "(may/must) (not) be done" and predictive phrases "be (not) + adjective", which are very commonly used in written English. The rules to adjust the

Table 3.2: Templates for extracting condition predicates

Templates
EX*/NN*/PRP*[a_1] MD*VB* (*not/no*)[R] (DT*JJ*)NN*/PRP*[a_2] (.)*
EX*/NN*/PRP* [a_1] MD*VB* (*not/no*) VB*[R] (.)*
EX*/NN*/PRP* [a_1] MD*VB* (*not/no*) (DT) NN*/JJ* [R] (.)*

extraction of object is a little complex since the composition of the object of a sentence is usually complex. Each dependency tree obtained from sentences in the main/alternative flow sections are analyzed with the adjusting rules. The accuracy of applying these rules in extracting action tuples is increased by more than 21% on the documents we used for evaluation. For subject and object extraction, the phrase structure tree of the same sentence is queried to identify modifying/subordinating information.

Extract Condition Predicates We notice that sentences which contain conditions are often complex, e.g., with sub-clauses. Therefore the dependency parser is likely to be less accurate, especially for the condition sub-clause. We also notice that the condition sub-clauses are usually written in simple formats. Therefore we extract the condition predicates through template matching. The matching is conducted in two phases. We first identify the condition-containing sub-clause by matching the sentence with condition indicators, e.g., "if, else, whether". Then the condition sub-clause is matched with the predefined templates shown in Table 3.2. The templates are defined based on POS tags. The label in the square bracket [] represents the corresponding field, i.e., the field R, a_1 and a_2, of the condition predicate (Definition 2). Words in the bracket are optional in the template. For example, to process the alternative flow sentence in Figure 2.2, our method first truncates the condition-containing sub-clause, i.e., "if there is no such constraint in the system". The sub-clause matches the first template in Table 3.2, and condition predicate $(2, \mathit{is\_no}, \mathit{there}, \mathit{constraint})$ is obtained.

Algorithm 1: Build Activity Diagram

Input : uc: a use case
Output: ad: an activity diagram

Initialize$(pnode, source, target)$
while *Enumerate* s *in* $uc.MF. \cup (uc.AF)$ **do**
  if s *is the last entry in* $uc.MF$ *or* $uc.AF$ **then**
    if *There is no fnode* **then**
      Build a final node *fnode*
    BuildEdge$(pnode, fnode, \epsilon)$
    continue
  Build an action node *anode* for s
  if s *is the first entry in* $uc.MF$ **then**
    Build an initial node *inode*
    $pnode \leftarrow inode$
  if $s.c \neq NULL$ **then**
    Build a decision node *dnode*
  if $s.n_s = NULL$ **then**
    $source \leftarrow pnode$
  else
    $source \leftarrow s.n_s$
  if $s.n_j = NULL$ **then**
    $target \leftarrow s$
  else
    $target \leftarrow s.n_j$
  if $s.c \neq NULL$ **then**
    BuildEdge$(source, dnode, \epsilon)$
    $source \leftarrow dnode$
  BuildEdge$(source, target, s.c)$
  $pnode \leftarrow anode$
  append all generated nodes to $ad.N$
  append all generated edges to $ad.E$
$ad \leftarrow AD(uc.UCName, uc.PreC, uc.PostC, ad.N, ad.E)$

3.4.4 Building Activity Diagram

When building activity diagrams, we are only interested in the sentences which describe real actions. However, there may be cases in which a sentence describes a constraint or an example. For example in Figure 2.2, the sentences in step 47 are not action steps. We noticed that most such sentences are either descriptive sentences which have modal verbs, such as "would, could, should, must, may" or illustrative sentences containing key words such as "for example, e.g.". Our method removes this kind of sentences through keyword matching. We found that the simple keyword matching achieves a good accuracy in filtering such irrelevant sentences.

Algorithm 1 shows the procedure of building an activity diagram from a use case structure. The main idea is to link action tuples based on control flow information. There are two kinds of control flow indicators. One is the control flow information, such as "go/jump to step #", that we identified by analyzing the content of a sentence. The other is the structure of use cases, i.e., consecutive sentences in each section of the use case represent the ordering in the control flow. Sentences in the alternative flow section are the branch flows of sentences in the main flow section.

Algorithm 1 takes a use case structure as input and produces an activity diagram. The use case name, precondition, postcondition fields of them are identical (line 29). The nodes and edges of an activity diagram are generated from the main flow and alternative flow sections of a use case. The variables *pnode, source, target* are all of type Activity Node (Definition 5). *pnode* represents the present node being processed; *source* and *target* are the identified source and target node of an edge. The code in line 3-13 creates nodes, including initial nodes (line 9-11), final nodes (line 3-7), action nodes (line 8) and decision nodes (line 12-13), of the activity diagram. The code in line 14-25 creates edges for the activity diagram. Line 14-17 identifies the source node of the edge. If the current sentence does not

have a specified starting node (line 14), the source node of the edge is set to the previous step (line 15) by default. Otherwise the source node is set to the specified starting node (line 17). Similarly, line 18-21 identifies the target node of the edge. If a choice node is created, an edge from the identified source node to the choice node should be created (line 22-24). For example in Figure 3.1, the edge from activity node $(42, (Check, \_, Order))$ to decision node d_{43} is build with the code in line 23. The edge from d_{43} to $(43, (Record, \_, value))$ is built with the code in line 25. All the created nodes and edges are appended to the node and edge field of the activity diagram ad (line 27-28). The ϵ field in the *BuildEdge* function (line 6, 23) represents there is no guard condition on the edge.

3.4.5 Formal Definition for Use Case Defects

To guide our analysis towards finding defects in use cases, we formally define the defects originally proposed by Töner et al. [121] and develop algorithms to systematically find them. Specifically, we focus on defects which are objective and have high defect intensity, such as missing elements, inconsistent step numbering, and misuse of preconditions. Defects, such as clarity of contents, level of details in the description, which are subjective are not considered. We focus on five types of defects, as defined below.

Inconsistent step numbering captures the situation where the sentence numbers of main flow or alternative flow are not consistent. This may lead to incorrect step referencing. For example in Figure 2.2, the step number 3 is missing in the main flow. As a result, the alternative flow has referred to a non-existing main flow step.

Definition 9 (D_1) *Given $uc \in UC$, if $\exists\, s, (uc.MF.cont(s) \vee uc.AF.cont(s)) : (s.n_s \neq NULL \wedge \neg\, uc.MF.cont(s.n_s) \wedge \neg\, uc.AF.cont(s.n_s)) \vee (s.n_j \neq NULL \wedge \neg\, uc.MF.cont(s.n_j) \wedge \neg\, uc.AF.cont(s.n_j))$, the use case is said to have inconsistent step numbering defect. The function cont() checks whether the item is a member of the list.*

In some use cases, the starting step (in main flows) of the alternative flow is not clearly specified. This may lead to ambiguity when merging the alternative flows with the main flow (to identify the overall flow information of the use case).

Definition 10 (D_2) *Given* $uc \in UC$, *if* $\exists\, s, uc.AF.cont(s) : s.n_s = NULL$, *then the use case contains the unclear alternative flow starting step defect.*

An overly-strong precondition is one such that inconsistencies between the precondition and the guard conditions of an edge may occur. For example in Figure 2.2, the sentence in the precondition has already restricted the order to be legal, thus the second sentence of the main flow, which checks the validity of the order, is redundant. If there is an alternative flow specifying the actions when the order is illegal, it is infeasible due to the precondition.

Definition 11 (D_3) *The precondition of an activity diagram ad is overly-strong if given an activity diagram* $ad \in AD$, $\forall\, prec \in ad.PreC, \exists\, e \in ad.AE : Conflict(e.g, prec)$, *where Conflict is a function deciding whether two predicates conflict as defined in Algorithm 3.*

Missing of alternative flows is the case when the main flow defines some action under some specific condition, however not all the other possible conditions are addressed. We regard this kind of situation as missing of alternative flows. For example in Figure 2.2, step 43 in the main flow specifies the condition "if the order is legal". But it is not specified what if that condition does not hold.

Definition 12 (D_4) *Given an activity diagram* $ad \in AD$, $\forall\, n \in ad.N$, *if* $n \in N_c \wedge n.t = decision \wedge OutG(n) = 1$. *The use case contains the missing alternative flow defect.* $OutG(n) \triangleq \mid e \in ad.E \wedge e.sn = n \mid$ *returns the number of edges out going from a given node.* $\mid A \mid$ *is the cardinal number operation on the set* A.

Algorithm 2: Check Unnecessary Strong Precondition

Input : ad: activity diagram
Output: whether find an over strong precondition or not

```
1  let n be the initial node
2  while There are unvisited nodes in ad do
3      mark n as visited
4      if n ∈ N_a then
5          if InChangeStatusDict(n.α.vb) then
6              return false
7      if n ∈ N_c ∧ n.t = decision then
8          if guard condition of any edges outgoing from n is Conflict with any
           p ∈ ad.PreC then
9              Report an over strong precondition defect
10             return true
11     set n to the next node following the edge in ad
12 return false
```

All the above types of defects are presented within one use case. There are also defects related to the inter-inconsistency among use cases. This may happen in use cases with inclusion relations, which require that the post-conditions of the included use case is comparable with the pre-conditions of the including use case.

Definition 13 (D_5) *Let* $uc_1, uc_2 \in UC$ *be two use cases. If* $\exists\, prec \in uc_1.PreC : prec.UC_{ref} == uc_2.UCName$, *then* $\forall\, postc \in uc_2.PostC \;\; \nexists\, prec \in uc_1.PreC : Conflict(prec, postc)$

3.4.6 Finding Defects

Following the definitions in Section 3.4.5, we discuss each defect finding method in details in this section. Notice that the focus of our work is not to find all possible defects (which is impossible). We are interested in developing methods that can tolerant the inaccurate natural language parsing results and detect highly likely defects.

Inconsistent step numbering (D_1) is checked based on the use case structure. We check all

the sentences in the MF and AF sections of a use case. If we find that an n_s field refers to a sentence that is neither in MF nor AF, an inconsistent step numbering defect is reported.

Unclear alternative flow starting step (D_2) is also detected based on the use case structure (Definition 4). We check all the sentences in the AF section of a use case. If we find a sentence with its n_s field not specified, an unclear alternative flow starting step defect is reported.

The process of detecting unnecessary strong preconditions (D_3) is shown in Algorithm 2. The rationale is that the pre-condition of a use case is the required initial status of the use case. If the status is not changed by the action steps, it should be preserved. The input to Algorithm 2 is an activity diagram ad. We traverse the activity diagram ad (starting from the initial node) in the while loop. For each node n, if it is an action node (line 4), we first check whether the action verb $n.\alpha.vb$ associated with the node is a status-changing verb (line 5). If yes, we stop and return false. Otherwise, if it is a decision node (line 7), we further check all the predicates associated with the edges outgoing from the decision node and see whether they conflict with any precondition predicate of ad (line 8). If yes, an over strong precondition defect is reported.

To decide whether a verb is status-changing, we manually defined a status-changing dictionary based on all the main verbs in the action tuples that we extract in step 3 (in Figure 3.2). For example, the action "Save the order" may change the status of the order and "check the order" will not change the order status. To avoid false positives, we only checks confliction-s between the pre-condition predicates and the predicate (associated with an edge) when there are no actions in-between that may change the status of the object. For example, in Figure 3.1, the edges from the first decision node d_{43} will be checked, since the action $(check, \_, order)$ will not change the status of order. But $(save, \_, order)$ is considered to be able to change the status of an order. Therefore any decision nodes after action node 45 in Figure 3.1 are not checked in our method.

Algorithm 3: Check Conflict Predicates

Input : p_1, p_2: two predicates
Output: whether the predicates conflict or not

```
1 if p1.ar ≠ p2.ar then
2     return false
3 if !Comparable(p1.R, p2.R) then
4     if p1.ar = 1 ∧ Sym(p1.a1, p2.a1) then
5         return true
6     if p1.ar = 2 ∧ Sym(p1.a1, p2.a1) ∧ Sym(p1.a2, p2.a2) then
7         return true
8 return false
```

Algorithm 3 shows the procedure of detecting conflicts on two predicates. To decide whether two predicates conflict, we first check whether the two predicates have the same arity (line 1). If yes, we check whether the predicate verbs are comparable or not (defined in function *Comparable*). If two verbs are comparable, the two predicates are comparable. Otherwise, we further check whether the arguments have the same parameters as indicated in the synonym dictionary (line $4-7$), the function Sym checks whether two words are synonym words. If not, the two predicates are decided to be comparable. Otherwise, we have found conflicting predicate verbs on the same objects and thus the two predicates are not comparable. There are two cases where two verbs are considered not comparable: (1) the verbs have the same semantic meaning but one with negative modifiers such as "no, not". (2) the verbs are in the contradict dictionary. Similar with deciding the status-changing verbs, we manually extracted two domain-specific dictionaries based on the action tuples extracted in step 3. The manually dictionary defining process is liberated and reinforced by referring to the WordNet [9] lexical database for English. Given a word, we first inspect the WordNet database to find all possible synonyms/contradicts of it. Then we conduct a set conjunction of the result provided by WordNet with all the verbs extracted form the document. These are all processed automatically. The only manual work is to check the results produced by the conjunction process and decide whether they are valid synonym/contradict words in our

context.

To detect missing alternative flows (D_4), we traverse each activity diagram to check all the decision nodes and see whether they have branch edges. If no branch edge is present for a decision node with guard conditions, a missing alternative flow error is reported. For example in Figure 3.1, the dashed lined edge emanating from decision node d_{43} is reported.

If two use cases have inclusion relation, we need to check the consistency issues among those use cases. The rationale is that the post-condition of the included use cases must be comparable with the pre-condition of the including use case. We check all the post-condition predicates of the included use case with all the pre-condition predicates of the including use case using Algorithm 3 to find inter-inconsistency defects of use cases (D_5).

3.4.7 Training Dependency Parser

ZPar provides an easy way to train a domain-adapted dependency parser. The users need to provide dependency trees, shown in Figure 3.3, as input. In our work, we randomly selected 18% of sentences in the stock trading system document, checked and labeled those sentences that are not correctly parsed (90 in the 314). These sentences are used to train a domain-adaptive parser for the stock trading system. The number of annotated sentences is less than 6% of the total sentences in the document. To preserve the generality of the trained parser, our training data set is merged with the Wall Street Journal sections of the Penn Treebank. We use the trained dependency parser to replace the original dependency parser in the process shown in Figure 3.2 and parsed the stock trading system again. The results are reported in Section 3.5.1.

Table 3.3: Use Case documents statistics

document type	# document	# use case	# sentence	ROOT accuracy
stock trading sys	1	188	1711	80%
academic sys	4	31	291	78.5%

3.5 Evaluation

We have implemented our approach in a prototype tool in Python. To test the applicability of our approach, we evaluate our methods with 219 use cases, which cover different application domains, ranging from financial, health care, machinery, monitoring and e-commerce systems. These use cases are adopted from a real industry system as well as academic publications. The statistics of the use cases that are used for our evaluation is shown in Table 3.3. The first row shows a use case document describing a real world stock trading system, with 188 use cases and 1711 sentences in total. This document is written by non-native English speakers. It contains many grammatical errors, which raise great challenge for our parsing. The second row is the statistics of 4 different system descriptions (i.e., a personalized health informatics system, an automated guided vehicle system, an emergency monitoring system and an online shopping system) from academic publications. There are 31 use cases with 291 sentences in total.

The experiments are conducted on a PC with Intel Core i7-2600 CPU at 3.4GHz and 4GB RAM. Due to the space constraints, we put all the experiment data (documents, activity diagrams and defect reports) and the implementation information (code, parser) on our website[4].

Since natural language parsing is highly sensitive to the writing styles of the sentences, in order to show clearly how effective our analysis algorithm is, we first check the root accuracy of ZPar on our documents, which in comparison with the root accuracy (90%) of ZPar on

[4] www.comp.nus.edu.sg/~lius87

the Wall Street Journal sections of the Penn Treebank dataset, acts as a measurement of the difference in writing styles between our document and the training set of the parser. The root accuracy of ZPar on the the stock trading system and the academic systems is 80% and 78.5% respectively, which reflect that our documents have different writing styles with the data set used to train the dependency parser.

We try to answer three questions with the evaluation.

- How accurate is the dependency parser in identifying action tuples and predicates? How much improvement can be achieved through our adjustment rules? How much improvement can be achieved through training a domain-adaptive parser?
- What is the accuracy of the activity diagram generation method?
- Is our defect finding method effective and accurate for automatic defect detection?

We remark that, we manually check the processing results to decide whether they are correct or not.

3.5.1 Accuracy of Free Text Parsing

The accuracy of the free text parsing is measured by the accuracy of the action tuples and predicates generated. Table 3.4 shows the evaluation results. The columns, from left to right, represent the document used ("doc"); the evaluation type ("type"); the number of wrongly ("# wrong"), partially correctly ("# PC") and correctly ("# correct") identified action/predicate; the total number of sentences ("# total") and the precision ("*prec*(*pprec*)") of the corresponding evaluation type. The precision is calculated by the formula $prec = \frac{\#correct}{\#total}$ and partial precision $pprec = \frac{\#correct+\#PC}{\#total}$. An action tuple is identified as correct if all the three fields of it are correctly extracted. It is identified as partially correct iff it

Table 3.4: Accuracy of parsing

doc	type	# wrong	# PC	# correct	# total	*prec*(*pprec*)
sts	Action(w/o)	155	22	137	314	43.6%(51.6%)
	Action(w)	82	23	209	314	66.6%(73.9%)
	Predicate	31	17	91	139	65.5%(77.7%)
as	Action(w/o)	105	52	131	291	45.0%(63.9%)
	Action(w)	69	14	208	291	71.5%(76.3%)
	Predicate	2	1	14	17	82.4%(88.2%)

has the main verb correctly identified and the subject/object incorrectly identified. We are interested in the partial precision because in some cases of defects detection, only the main verb information is critical. For example in Algorithm 3, only the verbs in the predicates need to be checked if they are not conflicting. The two rows "sts" and "as" represent the stock trading system and the academic systems respectively. Due to the large number of sentences in the stock trading system, we randomly sampled 18% of the sentences in the document to check the accuracy.

To show how robust/extensible the dependency parsing can be, we check the accuracy on identifying action tuples based only on the dependency label, i.e., ROOT for main verb, SUB for subject and OBJ for object. The results are shown in the "Action(w/o)" rows in Table 3.4. We can see from the results that, even without applying any of the adjusting rules, we can achieve more than 43% accuracy on identifying action tuples. The partial accuracy is even higher if we can tolerant an inaccurate object/subject information. The accuracy of the analysis results (Action(w)) with our provided adjusting rules is much higher. Only 1/4 are wrongly generated. For the stock trading system, 2/3 of the action tuples are correctly generated in the inspected sentences. The accuracy for the academic systems is higher (71.5%). Remind that the root accuracy of ZPar on our documents are around 80%. If we only consider those sentences which are assigned correct root by ZPar, our adjusting rules achieves an accuracy of 83.25% and 91.1% for the industry and academic systems respectively. This reinforce the claim that the rules we provided are general.

For the stock trading system, since it is written by non-native English speakers, there are many grammar errors, such as using noun and adjective words as verbs, which lead to the incorrect result. There are also very long sentences with complex attributive/adverbial clauses, which are usually colloquial and do not follow correct grammars. The dependency parser is confused by the wrong grammar and provides wrong parsing results which further lead to the wrong action tuples. The academic system documents are well written compared to the stock trading system, thus ZPar achieves a higher accuracy. Among the 69 sentences which have action tuples wrongly identified, 63 of them are because of the wrong ROOT labels. For the other 6 cases which have their root correctly labeled, the dependency relation between the real main verb and the root is not correct. Thus they are not correctly adjusted by our rules. For example, the sentence "The customer can either enter the account information. . . ", the modal verb 'can' is labeled as the ROOT of the sentence, however, the real verb 'enter' is not labeled to have dependency relation with 'can'. The main reason for incorrectly identifying objects is because the complex sentence structures, for instance, the object is a sub-sentence.

From the experiment results, we notice that (1) the rules (shown in Table 3.1) used to adjust extraction of action tuples are indeed useful. An increase in accuracy of 23% for the stock trading system and 26.5% for the those academic use cases is seen. (2) the rules are generalizable to different documents written by different development groups. It is promising to increase the accuracy by providing more rules.

We parsed the document with the trained parser, randomly sampled 18% of the results and manually checked the their correctness. Interestingly, 67 out of 310 have the root wrongly labeled, providing a precision of 78.4%, which is worse than the original parser. There are two possible reasons which may lead to this result. (1) The sentence writing style of the stock trading system is different from the Wall Street Journal dataset. (2) The wrongly labeled sentences contain grammar errors and proper nouns (like "GETS"). Therefore when

merging this dataset with the Wall Street Journal dataset, the parser is confused by such "inconsistencies".

3.5.2 Accuracy of the Activity Diagram Builder

We generate one UML activity diagram for each use case, the accuracy of the activity diagram building method is measured by the accuracy of the control flow information of the generated activity diagram. To be specific, we check (1) whether the nodes and edges that are correctly generated and linked in the activity diagram and (2) whether the guard conditions are correctly associated with the edges that are correctly generated in the activity diagrams.

For the stock trading system, 166 out of 188 use cases have correct nodes and edges generated. All the guard conditions are correctly associated with those edges. For the use cases which do not have activity diagrams correctly generated, 20 of them contain alternative flow steps which do not have clear starting steps; 8 of them have multiple conditions described within one step or in consecutive steps. Actually, this kind of writing style follows the If-statement style, which is not consistent with the majority of other use cases in the document, which follow the Cockburn style. For the academic use cases, 30 out of 31 use cases are correctly generated. The use case which is not correctly generated is because of missing of step information. This is caused by the over-strong filtering of irrelevant sentences.

3.5.3 Accuracy of the Defect Finder

To answer the third research question, we evaluate the accuracy of our defect finding methods. We adopt the standard metrics of precision (*prec*) and recall (*rec*) to evaluate the accuracy and effectiveness of our methods. We define $prec = \frac{|I_f \cap I_r|}{I_f}$ and $rec = \frac{|I_f \cap I_r|}{I_r}$, where I_f represents the set of items automatically identified by our method and I_r represents the

Table 3.5: Experiment results of defect detection

ID	I_f	I_r	$I_f \cap I_r$	*prec*	*rec*
D_1	18	18	18	100%	100%
D_2	22	20	20	90.9%	100%
D_3	19	21	19	100%	90.5%
D_4	83	59	59	71.08%	100%

set of defects that are manually detected from the document, which act as the baseline in the evaluation. We manually compare I_r with I_f to decide whether the automatically extracted item I_f is correct or not.

Table 3.5 shows the evaluation results on the stock trading system. There are 18 sentences from 11 use cases which are detected to have inconsistent step numbering problem (D_1). This result is identical with our manually checking result. 22 alternative flow steps are detected with defect type D_2, i.e., do not have clear starting step number. Compared to our manual detection results, 2 of them are false positives (i.e., correct but is identified as a wrong case). The reason is because of the irrelevant sentence presented in that step. Our method found 19 cases where the precondition is inconsistent with the guard conditions on the flow. Our manual detection finds 21 such cases. Therefore there are 2 cases that are missed by the automatic detection method. The reason is that our predicate extraction method fails to extract the correct predicates for the two cases. For detecting missing alternative flows (D_4), our tool found 83 potential defects presented in 39 use cases. 59 out of the 83 are real defects and 24 are false positives. The 24 false positives appear in 8 use cases, which have different writing styles with the majority of the other use cases. Thus our method failed to generate correct activity diagrams for those use cases, which further leads to those false positives. Although those false positives are not real missing alternative flow defects, they may cause potential maintenance problems, and thus are meaningful to be highlighted. Our tool did not report any inter-inconsistencies in the use case document. We did not find any such cases by manual detection either. Actually, the document is loosely

written such that use case preconditions and postconditions do not couple with each other well. Therefore there are very limited information we can use to do the checking. This is also the reason why the accuracy of the free text parsing does not affect too much on the accuracy of the defects detection.

The academic use case documents are well written and our tool successfully confirmed that there are no defects that we focus on.

3.6 Discussions

There are some limitations, manual efforts and threats to validity of our approach. We discuss them in this section.

Limitations (1) Currently, the action tuple definition captures the basic key elements, i.e., subject, object and main verb of a sentence. The purpose is to identify primary intentions of a sentence and enable automatic defects checking. For complex sentences which contain sub-sentences, there maybe multiple actions present in one sentence. Although this is not a recommended style of writing use cases [50], such cases may occur in a use case document. Currently the meaning of the sub-clauses is not further analyzed and thus there may be information loss. our method focuses on the main action of a sentence and may loose some information for those complex cases. We do , but information loss may still happen for complex sentences. For example, sentences like "Create two matches and send them to the trader; or create three matches and ...". The sentences indicates more than 3 actions. Currently we just extract one action tuple, i.e., $(create, two\_matches)$ from this sentence. This can be improved by extending the action extraction rule sets in Table 3.1.

(2) We only assign coarse semantic meanings, such as synonym/conflict/status-changing, to words in our method. This may lead to missing of cases in defect checking. For example, in

the over-strong precondition checking (D_3), our method only conducts the checking when we are sure that the actions in the flow do not modify the status of the object to be checked. However, some words are status-changing, but the resulting status does not conflict with the current status. Such cases will be missed by our method. For example, in Figure 3.1, the action $(save, \_, order)$ changes the status of the order, but does not affect the legality of the order in this case. However, our method will ignore all the branch conditions after the $(save, \_, order)$ action node, since the verb "save" indicates changes of status on order. Assigning fine-grained semantic meanings to words may solve this problem.

Manual Efforts In our approach, there are three steps which may require human intervention.

(1) If the input use case document does not follow the Cockburn writing style [39], some efforts of rewriting the use case document into the Cockburn style are needed in order to correctly generate activity diagrams. It is a common assumption in model transformation works [137] that the input model follows certain format.

(2) To decide conflict predicates, three domain-specific dictionaries, i.e., the synonym dictionary, the conflict dictionary and the status-changing verbs dictionary, need to be manually categorized. Our method provides all possible candidates for the dictionaries based on our automatically extracted subject, object and main verbs for each sentence. Then the WordNet lexical database is inspected to provide the preliminary synonym/conflict dictionaries. Therefore the only manual effort is to check and decide the dictionaries based on the preliminary dictionaries. Our method generated 262 distinct main verbs, among which 214 are valid, for the stock trading system document. With the aid of WordNet, we identified $20, 8$, and 14 groups of words for the synonym, conflict and status-changing dictionary respectively. The whole procedure takes a PhD student around 2 hours.

(3) If a user wants to train a domain-adaptive parser, manual efforts on labeling sentences

into dependency trees are required. The workload of this step depends on the number of sentences that are to be labeled. However, this step is optional in our approach. Our method achieves good accuracy without the training process.

Threats to validity There are some threats to validity in our evaluation of results.

(1) In the evaluation, we manually inspect each kind of defects and use the manual inspection results as the baseline. The manual defects detecting is subjective to the experimenter's understanding. To reduce this factor in our evaluation, the documents are checked by two PhD students in School of Computing, NUS, who have requirement engineering and natural language processing background.

(2) For the stock trading system, since it has more than 1700 sentences, we did not check all the sentences when evaluating the accuracy of free text parsing. To reduce the possible threats to validity caused by this, we randomly sampled 18% of the sentences and manually inspect the results.

3.7 Chapter Summary

In this chapter, we propose a method to automatically detect intra-defects in natural language use case descriptions. Our method leverages on the dependency parsing technique which allows document-independent rules to be provided. It is more adaptable to documents of different writing styles than template matching based on shallow parsing techniques. We proposed an algorithm to automatically generate an UML activity diagram, which captures the control flow information, for each use case. We formally defined common use case defects and defect detection techniques accordingly. The evaluation with 5 different use case documents shows that our method is effective in finding potential defects. Our method provides horizontal links to the original document to enable easy manual validation. The

defect report is presented in natural language, which greatly improved the involvement of stakeholders.

Chapter 4

Improve Use Case Document Quality Through Active Learning

Validating and maintaining a high quality use case document is crucial, which unfortunately is also subjective and labor-intensive, as we have discussed in chapter 3. One of the reasons why it is hard to have high quality use cases is that stakeholders usually do not describe the requirements clearly, consistently or completely [121]. Common problems with use cases include ambiguity in the description, inconsistency in the requirements and, perhaps more importantly, missing scenarios [53].

We have shown how to detect inconsistency defects and incompleteness related defects in a single use case in chapter 3. In this chapter we aim to develop methods and tools which aid finding potential missing scenarios and preconditions/postconditions that involve multiple use cases.

4.1 Introduction

As has been strengthened by Firesmith in his paper discussing "what makes good requirements" that "An entire requirements specification should be complete and contain all relevant requirements" [53]. However, the reality is that people usually "take certain information for granted and omit it, even though it is not obvious..." [53]. Through the working experiences with our industry collaborators, we have the following observations about requirements:

- Requirements specifiers usually have strong assumptions on the background/domain knowledge on the readers of the document. Sometimes they are just not careful enough to think thoroughly about all possible cases of the requirements. Therefore requirements are usually not completely specified.
- The incompleteness of requirements is especially obvious for use cases, which are scenario-based techniques to capture requirements.
- We cannot completely eliminate user interactions. Given that many problems of the use cases are caused by incomplete user requirements, and there is no better way to obtain the information other than interacting with the stakeholders, we shall not try to eliminate user interactions. Rather, we should ask only the right questions in a way which is easy to understand by the stakeholders so that they are not overwhelmed or confused.

Based on the above observations, we propose to improve the quality of use cases using techniques including natural language processing and machine learning. Central to our idea is to discover potential problems which would manifest during implementation and report the problems at the level of use cases so as to improve the quality of use cases.

Figure 4.1 shows the high-level workflow of our approach. Firstly, we adopt advanced natural language parsing techniques [140] to extract structured format from individual use

case written in English, from which we obtain information on behavior of each actor in the system in a particular scenario. Taking a system engineer point of view, next we attempt to answer the question on whether there would be a concise implementation of the system such that the requirements are satisfied. To do that, we need to, for each actor in the system, not only figure out the relationship between its behavior in different use cases but also check whether the behaviors in different use cases can be grouped into a meaningful and succinct implementation. For the former, we extract predicates from preconditions and postconditions of each use case and use those predicates as guidance to construct a use case relation graph. For the latter, we adopt active learning techniques from the machine learning community to incrementally learn a Deterministic Finite-state Automaton (DFA) from the behaviors in individual use cases. The use case relation graph is then used to compose the learned automata for every actor to obtain a plausible implementation for the actor. We remark that the requirement engineer and stakeholder are involved along the way in the process. For instance, we would automatically infer relationships between preconditions and postconditions of different use cases as much as possible. When ambiguity rises, we generate questions in English to consult the stakeholders, e.g., whether a certain precondition is satisfied by certain postcondition; or whether a behavior anticipated through learning (for instance, an implementation with a small number of states would probably allow this additional behavior) is indeed allowed but is missing from the current set of use cases.

In this way, we are able to elaborate the use cases interactively with the stakeholders to improve their quality, for instance, by reducing ambiguity in precondition and postcondition descriptions, or by identifying missing use cases. We remark while we attempt to synthesize a plausible DFA implementation for each actor in the system, it is not the goal of this work. Rather it is a way of identifying problems in use cases, which is a more realistic goal from our point of view. Nonetheless, some of the artifacts generated in our method could

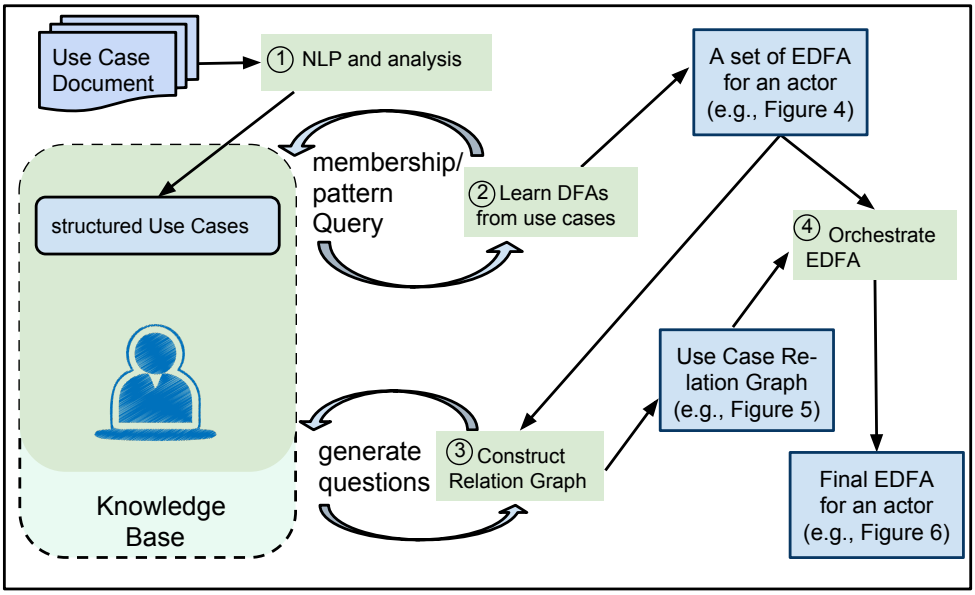


Figure 4.1: Overview of the quality improvement approach

be useful on its own. For instance, the use case relation graph shares the same utility as the use case charts or high-level Message Scequence Charts which are required as inputs of existing scenario based requirement validation approaches [126, 131]. Therefore, our approach can be used in combination with those approaches. We conduct a case study with a real industry use case specification document (of a financial system actively used by a financial institute in Boston), which contains more than 100 use cases describing the usage scenarios of 7 actors. We identified more than a dozen missing scenarios and dozens of other problems. The evaluation results show that our approach is effective in improving use cases with a reasonable amount of user interaction.

Outline. The remainders of the chapter are organized as follows. Section 4.2 illustrates our method with a running example. We provide background knowledge in Section 4.3. Section 4.4 discusses the details of our approach. In Section 4.5, we conduct a case study. We conclude this chapter in Section 4.6.

4.2 Running Example

In this section, we illustrate the overall process of our approach through an example. The example is adopted from our industry collaborator. For confidentiality, the use cases presented in this paper have been slightly modified, e.g., the sensitive words have been replaced. Nonetheless, the use cases remain largely faithful. Many of the use cases describe the interaction between a client and a server, with a range of operations such as creation, deletion, undo deletion. Figure 4.2 shows four sample use cases of the system, written in English. All these use cases describe the valid behavior of the actor "Ticker Monitor" and serve as input to our method.

There are four major steps in our approach. The first step is to "understand" the description of the use cases, i.e., we adopt natural language processing techniques to parse the natural language use case documents and obtain formal structures of the use case. In the second step, we formalize the behaviors of each actor in the use cases using DFA and make reasonable guesses on how the behaviors are to be realized. In particular, we adopt an active learning algorithm L^* [23] to learn an Extended Deterministic Finite Automaton (EDFA) which is compatible with the behaviors. This step allows us to improve the use cases through identifying missing scenarios. Different use cases might have very different preconditions and postconditions, in order to understand the relations between different use cases, we construct an use case relation graph (in Step 3) for each actor based on the preconditions and postconditions of each EDFA and then (in Step 4) compose the EDFAs to obtain an overall EDFA for each actor. This step allows us to reduce ambiguity in the use cases.

Step 1 : Natural Language Parsing We first adopt natural language processing techniques, i.e., dependency parsing and phrase structure parsing [140], to parse the sentences in a use case description into parse trees. Then we analyze the parse trees based on general

Use Case 1: Ticker Monitor Connects to GSYS
Initiating Actor: Ticker Monitor
Pre-Conditions
1. The ticker monitor can monitor the change of tick information and the connection status with TickFeed.
Main Flow
1. Ticker monitor connects to GSYS.
2. GSYS sends all Exch information and symbol information records in the database to ticker monitor.
3. Ticker monitor displays the records.
4. This ends the use case.
Alternative Flow
N/A
Post-Conditions
N/A

Use Case 2: Delete a Symbol Information from GSYS
Initiating Actor: Ticker Monitor
Pre-Conditions
1. The ticker monitor has connected to GSYS.
Main Flow
1. Ticker monitor selects a symbol information.
2. Ticker monitor sends a message to delete the symbol information.
3. This ends the use case.
Alternative Flow
N/A
Post-Conditions
1. The symbol information is deleted.

Use Case 3: Update a Symbol Information in GSYS
Initiating Actor: Ticker Monitor
Pre-Conditions
1. The ticker monitor has connected to GSYS.
Main Flow
1. Ticker monitor selects a symbol information.
2. Ticker monitor update some fields of the symbol information.
3. Ticker monitor sends the updated symbol information to GSYS
4. This ends the use case.
Alternative Flow
N/A
Post-Conditions
1. GSYS update the symbol information in the database if it is valid.

Use Case 4: Undo Delete a Symbol Information from GSYS
Initiating Actor: Ticker Monitor
Pre-Conditions
1. The ticker monitor has connected to GSYS.
2. GSYS has deleted one or more symbol information.
Main Flow
1. Ticker monitor undos delete the symbol information.
2. Ticker monitor sends the most recently deleted symbol information to GSYS.
3. This ends the use case.
Alternative Flow
1. In step 1, if ticker monitor has not deleted any symbol information, do nothing.
Post-Conditions
1. GSYS restore the symbol information.

Figure 4.2: Sample use cases

grammar rules that are extracted from the documents (refer to Section 3.4.3). We identify all the actions which are related to the actor of concern based on the parsed action tuples. For example, in use case 2 of Figure 4.2, the action tuples for the main flow sentences are *selects(Ticker monitor, symbol information), sends(Ticker monitor, message to delete symbol information)*. Both action tuples have actor "ticker monitor" as subject. We thus consider all of them as actions related to the actor. We utilize the same natural language parsing and analysis techniques with that we introduced in Chapter 3.

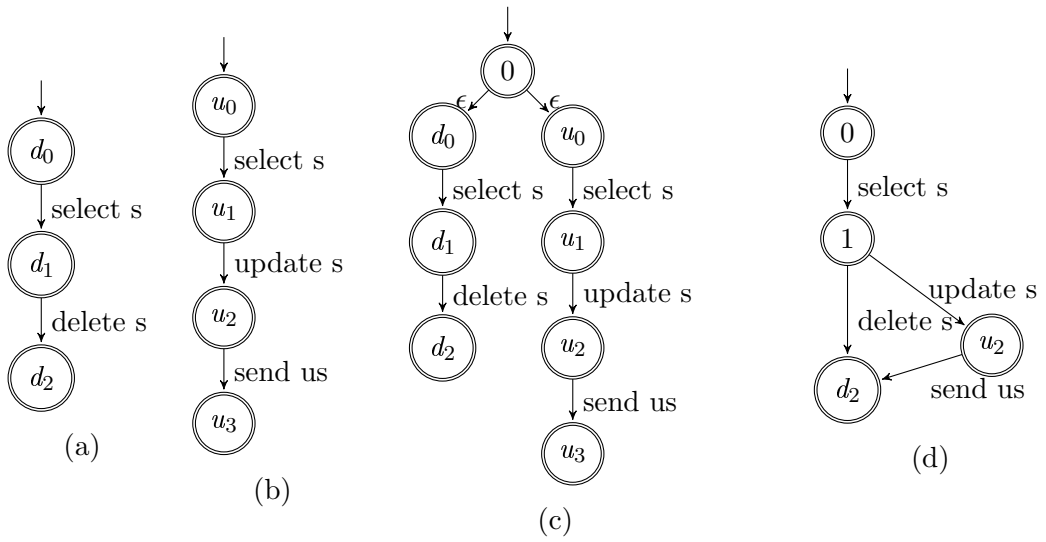


Figure 4.3: (a) The NFA for use case 2 in Figure 4.2; (b) use case 3 in Figure 4.2; (c) the merged NFA; (d) the corresponding DFA

Then the structured sentences are linked based on the control flow information (e.g., previous, succeeding relation or the "go to" statement) described in the flow steps to obtain a raw Nondeterministic Finite State Automaton (NFA), which captures the actions of one actor described in the use case. For example the NFAs[1] shown in Figure 4.3 (a) and Figure 4.3 (b) are constructed from use case 2 and use case 3 in Figure 4.2, respectively. We merge all those NFAs which describe the actions of the same actor and share the same preconditions to obtain one NFA. The NFAs in Figure 4.3 (a) and Figure 4.3 (b) are merged to obtain the NFA shown in Figure 4.3 (c). Then we determinize the NFA in Figure 4.3 (c) to obtain a DFA[2] shown in Figure 4.3 (d), which serves as a part of the knowledge base during the active learning process. In our approach, we learn the DFA which is prefix-closed with the assumption that the system can stay in any of the status after conducting an action. Therefore, we set all states in the obtained DFA to be accepting states.

[1]We simplify the action tuple representation to save space.

[2]We only show the transitions which lead to accepting states in the DFAs for clarity. The same applies to Figure 4.4 and Figure 4.6.

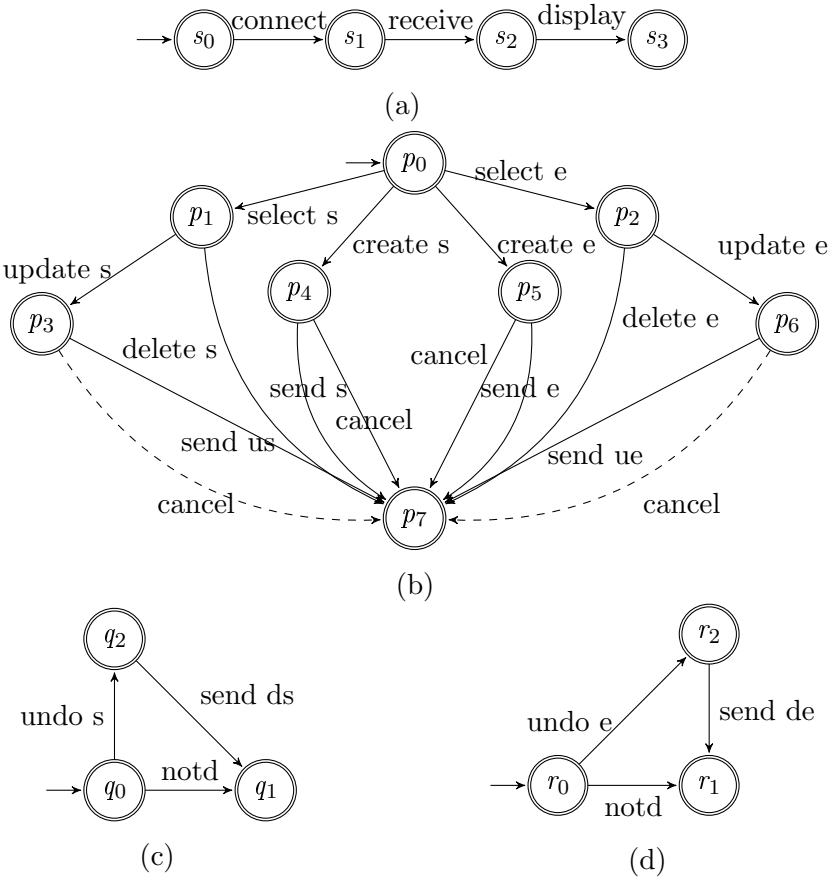


Figure 4.4: The partial DFAs for Ticket Monitor

Step 2 : Learn Local EDFA We group those structured use cases based on their preconditions and postconditions. For all the use cases which describe the actions of one actor, if they have the same preconditions, they are put together as one group (e.g., use case 2 and use case 3 in Figure 4.2). All the actions appear in those use cases in the same group are fed to the L^* algorithm as the alphabet to learn one local DFA. For example, in Figure 4.2, use case 1 and use case 4 correspond to DFAs shown in Figure 4.4 (a) and Figure 4.4 (c) respectively. The DFA in Figure 4.4 (b) is generated based on the traces from use case 2, use case 3 and some other use cases (we do not show them in Figure 4.2 due to the

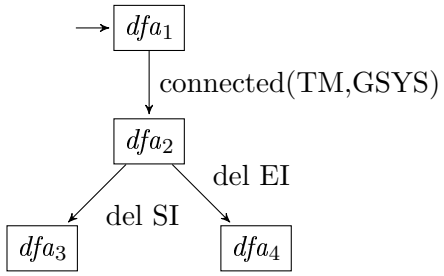


Figure 4.5: Relation graph of Ticket Monitor EDFAs

space limit) within the same group. We defer the detailed process to learn the DFA in Figure 4.4 (c) to Section 4.3 when introducing the L^* algorithm. One goal of the learning is to gradually discover missing scenarios by generating questions to users. Using an active learning algorithm allows to 'control' the number of questions required. For instance, the dashed lines in the DFA shown in Figure 4.4 (b) represent the traces that are added during the interactive learning process. These traces are generated by our learning algorithm and are confirmed to be valid by users.

We then assign each DFA with preconditions and postconditions of the use cases that compose it. The DFA with preconditions and postconditions is called an Extended DFA (EDFA, refer to Definition 17). The postconditions of an accepting state are set on a trace basis, i.e., only the accepting states, e.g., p_7 in Figure 4.4 (b), which correspond to ending of traces have postconditions. For the trace $\langle$*select s, delete s*$\rangle$, the postcondition is set to be *deleted(symbol informtion)*. The trace-based preconditions and postconditions are used for the later use case relation graph generation and EDFA composition, when we decide how to split the traces.

Step 3 : Construct Use Case Relation Graphs The relations such as "happen before" or "in parallel" usually exist for use cases of one actor. Those relations can be inferred from the precondition and postcondition sections in the use case documentation. On the

other hand, those relations can be used to identify ambiguity in precondition/postcondition descriptions as well. Recall that in step 2, we learn one EDFA for use cases with the same precondition. Therefore usually multiple EDFAs are learned for one actor. In order to show the overall view of all the valid behaviors of an actor, we build a usage relation graph for those EDFAs based on their corresponding preconditions and postconditions. The usage relation graph for the EDFAs in Figure 4.4 is shown in Figure 4.5. The nodes represent the EDFAs and directed edges represent the precedence of usage relations between two EDFAs. The labels on the edges are the conditions which the edge linkages are based on. For example, dfa_3 "happens after" dfa_2 based on the common condition *the symbol information is deleted* (represented by *del SI* in Figure 4.5), which is the postcondition of dfa_2 and the precondition of dfa_3. Likewise, dfa_2 and dfa_4 are linked based on the common condition *the Exch information is deleted* (*del EI* in Figure 4.5).

We construct one usage relation graph for each actor. For those use cases with trivial linking conditions, we link them directly. For example dfa_2 and dfa_3 can be linked directly based on the predicate *del SI*. For those use cases which do not have clear linking references, or miss preconditions/postconditions, we raise natural language questions to query the users about the relations between those use cases. For example, the EDFA in Figure 4.4 (a) (corresponds to use case 1 in Figure 4.2) does not have postconditions specified. Our method raises questions based on the preconditions of existing use cases for users' confirmation. In this example, the precondition *connected(TM, GSYS)* ("*The ticker monitor has connected to GSYS*"), from all the existing use cases, is confirmed to be a legal postcondition for use case 1.

Step 4 : Orchestrate Local EDFAs Afterwards, we orchestrate all the EDFAs based on the usage relation graphs obtained in step 3. We traverse the usage relation graph in a breadth-first manner and link a node with all its child nodes on the common conditions

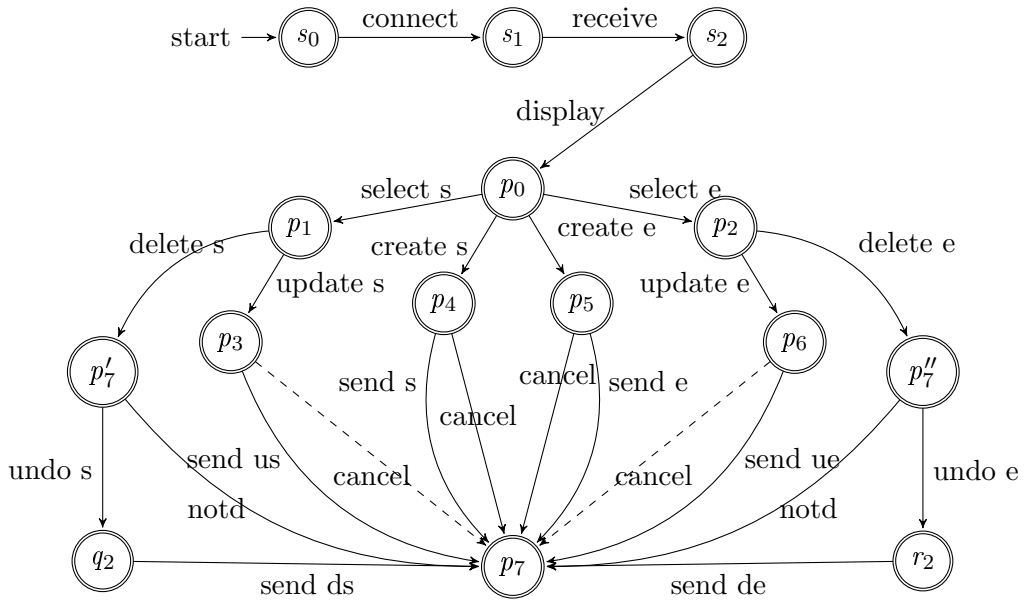


Figure 4.6: The overall DFA for Ticket Monitor

labeled on the corresponding edges. For example, according to the relation graph shown in Figure 4.5, dfa_1 (Figure 4.4 (a)) and dfa_2 (Figure 4.4 (b)) are linked based on the common condition *connected(TM, GSYS)*, which is the precondition of dfa_2 and the post-condition of dfa_1.

There are cases where we need to split a final state during the orchestration. Since each trace has a set of corresponding postconditions, those traces which have comparable post-conditions with the preconditions of a given EDFA are split out to link with the initial state of that EDFA. For example, according to the graph shown in Figure 4.5, dfa_2 links with dfa_3 on the common condition *del SI* and link with dfa_4 on the common condition *del EI*. The result of the orchestration is shown in Figure 4.6. Traces $\langle$*select s, delete s*$\rangle$ is split out to link with dfa_3 based on the common condition *del SI*. Similarly traces $\langle$*select e, delete e*$\rangle$ is split out to link with dfa_4. The orchestrated DFA is shown in Figure 4.6.

As illustrated in the running example, our approach is able to find missing scenarios, e.g.,

the traces indicated in dashed lines in Figure 4.6, as well as missing preconditions/post-conditions, e.g., the postcondition of use case 1 in Figure 4.2, through active learning and interaction with stakeholders.

4.3 Preliminary

In this section, we present preliminaries on active learning and the L^* algorithm that we adopt in our approach.

Active learning refers to a model of instruction in which a student interacts with a teacher by actively asking questions in order to learn the knowledge. Angluin proposed the L^* algorithm [23] to learn the unknown DFA U (i.e., the knowledge) from the teacher, who knows the DFA, by asking *membership queries* and *candidate queries*. For a membership query, L^* asks the teacher whether a string s is a member of the accepted languages of the unknown DFA, i.e., whether s is accepted by U. The teacher answers yes(1)/no(0) accordingly. After a set of membership queries, L^* conjectures a candidate DFA C from his current knowledge and asks the teacher a candidate query whether the candidate DFA is equivalent to the unknown DFA, i.e., $C \equiv U$. If the teacher answers *yes*, L^* successfully learned the DFA, which is equivalent to the current candidate DFA; If the teacher answers *no*, it provides a counterexample trace which is either accepted by C or U but not both. L^* then extracts knowledge contained in the counterexample and starts asking membership queries. L^* is guaranteed to terminate and learns U within polynomial time (see the proof in Angluin's original paper [23]).

L^* represents its current knowledge with an *observation table* $K = (S_K, E_K, T_K)$, in which S_K is a set of row index strings and E_K is a set of column index strings. T_K is a mapping: $S_K \times E_K \rightarrow \{yes, no\}$, which contains the knowledge whether a string $s \cdot e$ concatenated from $s \in S_K$ and $e \in E_K$ is accepted by U or not. In the following we abuse notation $T_K(s)$

to denote a row of | E_K | number of *yes*s or *no*s corresponding to a row index string $s \in S_K$.

L^* assumes the alphabet Σ of U is known beforehand. Both S_K and E_K are initialized with the empty string λ. For each symbol $a \in \Sigma$, $s \in S_K$, and $e \in E_K$, if $s \cdot a \notin S_K$, it asks the teacher a membership query whether the string $s \cdot a \cdot e$ is accepted by U. If the string is accepted by U it maps $T_K(s \cdot a, e) \rightarrow \{yes\}$; it maps $T_K(s \cdot a, e) \rightarrow \{no\}$ otherwise. L^* asks membership queries until K is *consistent* and *closed*, i.e., for any $s \in S_K$ the row $T_K(s)$ must appear more than once. If K is not closed, let $s \in S_K$ be a string for which $T_K(s)$ appears only once, then for each $a \in \Sigma$ and each $e \in E_K$, it asks a membership query for string $s \cdot a \cdot e$. When K is consistent and closed, L^* conjectures a candidate DFA C from K as the following:

- $S(C)$ are the distinct rows $T_K(s)$ for all $s \in S_K$.
- $init(C)$ is the row corresponding to $T_K(\lambda)$.
- $AS(C)$ are the rows $T_K(s)$ for which $T_K(s, \lambda) \rightarrow yes$.
- $\delta(C)$ is defined as for $s_i, s_j \in S(C)$, $\delta(s_i, a) = s_j$ if there is a $T_K(s) = s_i$ and $T_K(s \cdot a) = s_j$.

We adopt the Rivest and Schapire approach [113] to handle a counterexample string ν returned by the teacher when $C \not\equiv U$. First we find the longest prefix $u \in S_K$ of ν, such that $\nu = u \cdot v$ and add all suffixes of v which do not exist in E_K into E_K. Then we fill up T_K by asking membership query for each string concatenated with each $s \in S_K$ and each string in suffixes of v, and then check whether K is closed.

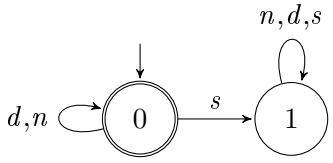
We illustrate how L^* works to learn the DFA in Figure 4.4 (c). To simplify the presentation, we use the symbol d, s, n to represent the alphabet symbol *undo s*, *send ds* and *notd*, respectively. At the beginning, the observation table is shown in Figure 4.7 (a). This table is

	λ
λ	1
d	1
n	1
s	0

(a)

	λ
λ	1
s	0
d	1
n	1
sd	0
sn	0
ss	0

(b)



(c)

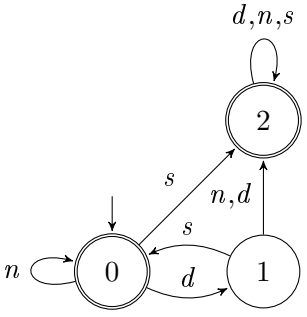
Figure 4.7: The observation tables (a) and (b) in the first learning round and the first candidate DFA (c)

	λ	s
λ	1	0
s	0	0
d	1	1
n	0	0
sd	0	0
sn	0	0
ss	0	0

(a)

	λ	s
λ	1	0
s	0	0
d	1	1
n	1	0
sd	0	0
sn	0	0
ss	0	0
dd	0	0
dn	0	0
ds	1	0

(b)



(c)

Figure 4.8: The observation tables (a) and (b) in the second learning round and the second candidate DFA (c)

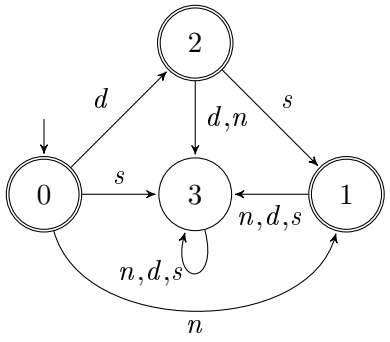
not closed because row indexed by string s appears only once in the table. L^* moves this row to the upper part and extends the table with each alphabet symbol by asking membership queries for strings sd, sn and ss. The extended table is shown in Figure 4.7 (b) which is closed. The first candidate DFA constructed from the table is shown in Figure 4.7 (c). Then L^* asks a candidate query with the candidate DFA, for which the teacher returns a counterexample string ds. The table contains string d which is the maximum prefix of ds. Thus the suffixes of string s are $\{\lambda, s\}$. Only string s is added to the table column because λ already exists in the column.

	λ	s	n
λ	1	0	1
s	0	0	0
d	1	1	0
n	1	0	0
sd	0	0	0
sn	0	0	0
ss	0	0	0
dd	0	0	0
dn	0	0	0
ds	1	0	0

(a)

	λ	s	n
λ	1	0	1
s	0	0	0
d	1	1	0
n	1	0	0
sd	0	0	0
sn	0	0	0
ss	0	0	0
dd	0	0	0
dn	0	0	0
ds	1	0	0
nd	0	0	0
nn	0	0	0
ns	0	0	0

(b)



(c)

Figure 4.9: The observation tables (a) and (b) in third learning round and the third candidate DFA (c)

Then L^* asks several membership queries to fill up the cells due to the addition of the column s. The observation table is shown in Figure 4.8 (a). This table is not closed because the row with valuation 11 appears only once in. L^* moves the row indexed by string d (the first row corresponding to row 11) to the upper part and extends the table with rows indexed by string dd,dn and ds by membership queries. The extended table is shown in Figure 4.8 (b), which is closed. Then L^* constructs the second candidate DFA shown in Figure 4.8 (c). For this candidate query, the teacher returns a counterexample string nn. Then the string n is added to the table column. L^* repeats the previous steps to obtain the third candidate DFA shown in Figure 4.9 (c) and asks a candidate query. The teacher finds that the candidate DFA is equivalent to the DFA to be leaned. Thus L^* successfully learns the DFA.

4.4 Detailed Approach

In this section, we present the details of the steps in our approach.

4.4.1 Natural Language Parsing and Analysis

Recall that as we have discussed in Section 2.2, there is no standard template for writing use case documents as concluded by Fowler [54]. We thus focus on one of the widely used writing styles in the literature and practice, which is "the single-column, numbered, plain text, full sentence form" [39]. Since the input of our approach is a use case specification document written in English, we adopt advanced natural language parsing techniques, i.e., dependency parsing and phrase structure parsing [140] to parse the document. Then we analyze the parse trees by rule matching to extract action tuples and predicates from the flow steps and precondition/postcondition sections. The formal definition of an action tuple and a predicate are defined in Definition 1 and Definition 2, respectively. After analyzing the parse trees, each sentence in the use case description is mapped into the formal structure as defined in Definition 3. The use case is organized based on the sections to which those sentences belong, as is formally defined in Definition 4. The detailed steps of analyzing parse trees have been discussed in Section 3.4.3 and thus we skip the details here.

Then an NFA is generated based on the control flow information extracted from the use case. NFA is defined as the following.

Definition 14 (NFA) *An NFA is defined as* $NFA \triangleq \{S, \Sigma, \delta, init, AS\}$*, in which* S *is a non-empty finite set of states;* Σ *is a non-empty finite set of alphabet;* $\delta = S \times \Sigma \to \mathbb{P}S$ *is a transition relations;* $init \in S$ *is the initial state and* $AS \subseteq S$ *is the set of accepting states.*

A Deterministic Finite Automaton (DFA) is a special NFA where there is no ϵ in the alphabet and the transition relation of a DFA is $\delta = S \times \Sigma \to S$. For example, in Figure 4.4 (a), the DFA can be represented as $\{\{s_0, s_1, s_2, s_3\}$,{connect, receive,display},$\{s_0 \xrightarrow{connect} s_1, s_1 \xrightarrow{receive} s_2, s_2 \xrightarrow{display} s_3\}, s_0, \{s_0, s_1, s_2, s_3\}\}$ by definition. DFA and NFA are proved to have the same power of capturing regular languages and there is standard powerset construction algorithms to construct an equivalent DFA from an NFA (see [110]).

Recall that as we have discussed in Section 3.4.4, there are two kinds of control flow information in a use case description, as is exemplified in use case 4 in Figure 4.2. The first kind of control flow information is represented by the sequential ordering of sentences in each section, i.e., the $s\#$ field in a sentenc S. The second kind of control flow information is represented by the conditional statement, which are usually indicated by the keywords such as "if", "whether", "else", in a sentence. The control flow information is extracted during the procedure of analyzing the parse trees (Section 3.4.3).

We then compose an NFA from a use case defined in Definition 4 following the steps shown in Algorithm 4. We first create a state for each sentence (line 1-3). Then we link the states based on the previous (n_s) and succeeding fields (n_j) of the corresponding sentences (line 4-11). If we find the sentence step number $s_i.s\#$ is the same with the succeeding node step number $s_k.n_j$ of another sentence; or the previous node step number of the sentence ($s_i.n_s$) is the same with another sentence step number $s_k.s\#$. then we add a transition between the corresponding nodes accordingly (line 6-7)). All the actions labeled with transitions are added into the NFA alphabet (line 8)). If the action field is empty, we label the transition with ϵ. Lastly, we set the initial state and accepting states based on the sequence of the sentences in the $UC.MF$ section. The state corresponding to the first sentence in $UC.MF$ is set as the initial state, and the states corresponding to the last sentences in $UC.MF$ and $UC.AF$ are set as the accepting states.

After obtaining an NFA for each use case, we construct an NFA for all the use cases sharing common preconditions, by adding one unique initial state and an ϵ transition from it to each initial state of the NFAs for those use cases. Then we convert the NFA to an equivalent DFA with the anti-chain improved powerset construction algorithm [133], which is often efficient despite its worst-case exponential complexity.

After this step, we obtain a set of DFAs for each actor. Each DFA corresponds to a set of use cases which have the same preconditions. Those DFAs serves as part of the knowledge

Algorithm 4: Generate an NFA from a Structured Use Case

Input : UC: a use case
Output: nfa: an NFA

```
1  for each s ∈ UC.MF ∪ UC.AF do
2      s ←create a state for s
3      nfa.S.add(s)
4  for each pair of states (s_i,s_k) from nfa.S do
5      Let s_i and s_k be the corresponding sentences
6      if s_i.s# = s_k.n_j or s_k.s# = s_i.n_s then
7          nfa.δ.add(s_k,s_i,s_k.α)
8          nfa.Σ.add(s_k.α)
9      else if s_k.s# = s_i.n_j or s_i.s# = s_k.n_s then
10         nfa.δ.add(s_i,s_k,s_i.α)
11         nfa.Σ.add(s_i.α)
12 nfa.Σ.add(ϵ)
13 set nfa.init and nfa.AS
14 return nfa
```

base in our learning process.

4.4.2 Learn the DFAs

In the second step, we adopt the L^* algorithm to learn a DFA representation of the actor's behavior, which we take as a hint on how the system can be possibly implemented. In our setting, the teacher is composed of the structured use cases (i.e., the DFA we obtained in step 1) and the user interacting with our tool. The alphabet is set as all the actions extracted from the structured use cases.

L^* is an active learning algorithm. Compared with the passive learning techniques, it has the advantage of obtaining new knowledge incrementally from beyond the given set of traces, and thus enables finding missing use cases interactively. In our approach, the membership query is answered based on both the known traces from the use cases as well as suggestions from the users. The candidate query is decided by checking the equivalence of two DFAs,

i.e., the DFA learned with L^* algorithm and the DFA we generated from the knowledge base (in step 1).

4.4.2.1 Membership Query

The membership query checks whether a trace generated by the L^* is a valid trace. To answer the query, we first check whether the given trace is a valid trace in the DFA obtained from step 1 by a depth-first search. If it is not, we raise a question to the user to ask whether the trace should be accepted or not. If the user answers "no", we reject the trace. Otherwise, we accept the trace and add the trace to the DFA generated in step 1.

The number of membership queries generated by L^* is linear to the size of alphabet, states in the DFA as well as the length of the returned counterexample, and thus may be rather large. We propose five filtering techniques to filter out those traces which are unlikely to be valid so as to reduce the amount of user interactions required.

The first filter is that we only allow the traces that start with actions which are initial actions of some known valid traces. For example, in the DFA of Figure 4.4 (c), only traces start with actions in the set $\{undo\ s,\ notd\}$ are raised as queries to the users. The justification is these initial actions are often 'special' and it is unlikely that the user would completely forget certain functionality of the system.

The second filtering technique is proposed based on the premise that the learned DFA is prefix closed. Therefore we can conclude that if a prefix of a trace is not accepted, the trace is not accepted. Thus we record all the traces that are denied by users for checking in later rounds.

The third filtering technique is based on the conflicting actions extracted from use cases. Actions or predicates with conflicting semantics are unlikely to reside in the same trace. For example in use case 4 of Figure 4.2, the action "undo delete symbol information" in step 1 of

the main flow and the predicate "not deleted (symbol information)" from the alternative flow cannot reside in one trace since they have conflict semantics. We extract these conflicting action/predicate pairs automatically from the use cases. Users can also provide conflicting action/predicate pairs.

The fourth filtering technique is based on the precedence order between actions, as is formally defined in Definition 16.

Definition 15 (Trace) *A trace is defined as* $T \triangleq\{ < A >, PostC\}$ *where PostC is the postconditions of the trace and* $< A >$ *is a list of actions.*

Definition 16 (Precedence Order) *Two actions* α_1 *and* α_2 *are said to have a precedence order relation* $\alpha_1 \prec \alpha_2$ *iff* $\exists\, t \in T : \alpha_1 \in t. < A > \wedge\, \alpha_2 \in t. < A > \wedge\, index(\alpha_1) < index(\alpha_2)$*, where the function index(*α*) returns the index of an action in the trace.*

The precedence order is a partial order relation, which satisfies the transitive property, i.e., if we have two pairs of actions $\alpha_1 \prec \alpha_2$ and $\alpha_2 \prec \alpha_3$, both of which satisfy the precedence order, then we can infer that the precedence order $\alpha_1 \prec \alpha_3$ holds. In practice, events in a systems are often ordered, for instance, login often precedes authenticate. Knowing the ordering between different events would help to greatly reduce the number of user interactions needed. In theory, only the user knows the ordering. Nonetheless, in our work, we infer likely ordering based on the given use cases and only in the presence of conflicts, we would consult the user.

We generate precedence order relations between actions in the alphabet based on their sequential ordering in the use case flows. For example in Figure 4.4 (a), we can obtain the partial order relation *connect*$\prec$*receive* and *receive*$\prec$*display* directly. We can infer the partial order relation *connect*$\prec$*display* from the two known partial order relations based on the transitivity property. If, however we get two conflicting partial orders through either direct

generation or inference, we may have encountered two possible situations. One situation is that we have encountered a loop and the two actions are involved in the loop. We then detect the loop and remove all the partial order relations involved in the loop. The other situation is that we found a conflict in the action execution sequence. This can be raised as a question for user decision. Any trace that contains a pair of actions which violates the valid partial order relations is denied without querying the users.

All the previous filtering techniques are based on the static information extracted from the use case descriptions. The last filtering technique is based on the answers provided by users on the fly. During the process of membership query, we raise questions (in the form of a sequence of actions) to users and users response with "yes" or "no". From the sequences that are rejected by users, i.e., the sequences that users answer "no", we mine frequent patterns with the Apriori algorithm [20]. We present the mined patterns to users for confirmations as to whether such sequences which contain those patterns are always not accepted. Those patterns that are confirmed by users are used in the membership queries for filtering.

All these filtering techniques are proposed based on the assumption that the valid traces in the use cases should share common features, such as the partial ordering between actions, common prefix, which can be mined from the existing knowledge. With all these filtering techniques, the number of questions generated is often controlled in a reasonable amount.

If a trace generated in a membership query is confirmed to be valid by users, we merge the valid trace with the existing DFA (generated from the knowledge base in Step 1). To do so, we find the state which shares the longest prefix with the given trace t from the initial state of the DFA by a depth-first search on the DFA. Then we add a sequence of transitions which capture the suffix of t from the current state of DFA. The obtained DFA is not guaranteed to be minimal, but is guaranteed to be deterministic. By interacting with the users, we are able to find missing scenarios which are not captured by the use case documents.

Algorithm 5: Candidate Query

Input : dfa_a: the dfa that is generated by L^*
dfa_b: the dfa constructed from the knowledge base
Output: A counterexample trace

```
1  if dfa_a.AS.isEmpty() then
2      return any accepting trace in dfa_b
3  construct the product P_× of dfa_a and dfa_b
4  while traverse P_× do
5      let (s_a, s_b) be a state in P_×
6      if s_a ∈ dfa_a.AS && s_b ∉ dfa_b.AS or
7      s_a ∉ dfa_a.AS && s_b ∈ dfa_b.AS then
8          return counterexample(trace,P_×)
9      trace.add(s_a, s_b)
10 return an empty trace
```

4.4.2.2 Candidate Query

To conduct candidate query, we check the equivalence of two DFAs, i.e, the DFA which is learned by the L^* algorithm and the DFA that is constructed from the knowledge base (in step 1).

To check the equivalence of two DFAs, we construct a product $P_\times$ of the two DFAs and check whether the accepting states of the product DFA are composed of pairs of accepting states from the two DFAs. If we find any state in $P_\times$ which is composed an accepting state from one DFA and a non-accepting state from the other, the two DFAs are not equivalent. This is easy to prove since we have found a trace which is accepted by one DFA but not the other. If the two DFAs are not equivalent, we need to return a counterexample to the L^* algorithm for refinement.

The algorithm to answer candidate queries is shown in Algorithm 5. We first check whether the DFA learned by L^* (dfa_a) has accepting states (line 1–2). If not, we pick one trace ending in an accepting state from the DFA constructed from the knowledge base (dfa_b) and return it as a counterexample. Otherwise, we construct a product DFA $P_\times$ of dfa_a and dfa_b

(line 3). We traverse the product DFA ($P_\times$) from its initial state (line 4–9). If we find a state pair (s_a, s_b) in the product ($P_\times$) such that s_a is an accepting state in dfa_a while s_b is not an accepting state of dfa_b; or s_b is an accepting state in dfa_b while s_a is not an accepting state of dfa_a, then a counterexample, which is the traversed trace in $P_\times$, is returned (line 8). Otherwise we record the state pair (s_a, s_b) (line 9). If we do not find any counterexample, an empty trace (line 10), which marks that the two DFAs are equal, is returned.

4.4.2.3 Set Precondition and Postcondition for EDFA

To enable the further orchestration of DFAs, we set preconditions and postconditions for those DFAs learned by L^* algorithm to obtain an Extended DFA (EDFA), which contains preconditions and postconditions, as is formally defined in Definition 17.

Definition 17 (Extended DFA) *An Extended DFA is defined as EDFA*$\triangleq$ *{DFA, PreC, PostM}, where DFA is a DFA; PreC* $\subset P$ *is the set of preconditions of the DFA; PostM is a mapping from each final state in DFA to their trace based postconditions* $\hat{T} \subset T$ *that compose this DFA.*

The precondition of a DFA is set as the union of the precondition sets of all the traces which are used to generate the DFA. Recall that in our approach, all the use cases for an actor which have the same preconditions are grouped to learn one DFA. Thus the precondition of a DFA is identical to the precondition of use cases from which it is learned. The postconditions of accepting states in the DFA are set to be the set of postconditions of the traces reaching the accepting state. Note that the postconditions are set on a trace basis, meaning each trace ending in the same accepting state has its own postconditions. For example, in Figure 4.4 (b), only state p_7 has postconditions and it is set as the set {⟨⟨*select s, update s, send us*⟩,*update(GSYS, symbol information)*⟩, ⟨⟨*select s, delete us*⟩,*deleted(symbol*

information)⟩,...}. The first two elements in the set represent the left-most two traces in Figure 4.4 (b). We omit the postconditions for the other traces for confidentiality. Assigning preconditions and postconditions correctly is critical for the splitting of traces during the orchestration of DFAs.

4.4.3 Construct Relation Graphs

We first construct a usage relation graph based on the preconditions and postconditions of EDFAs learned for an actor. A node in a usage relation graph represents an EDFA which is learned with the L^* algorithm. Initially, we have a set of nodes which represent the EDFAs learned for a set of use cases describing the usage scenarios of one actor. We link two EDFAs if the postconditions of an accepting state of an EDFA are comparable the precondition of the other EDFA. This case represents a succession in behavior. The relation graph is formally defined in Definition 18.

Definition 18 (Relation Graph) *A relation graph is defined as* $G \triangleq \{N, n_i\}$. *where* $n_i \in N$ *is the initial node of the graph and* N *is the set of nodes in the graph. Each node in the graph is defined as* $n \triangleq \{PreC, PostM, EDFA, cc, ch\}$, *where* $PreC$ *and* $PostM$ *are the precondition and postconditions of the EDFA the node represents. EDFA is the EDFA that the current node represents. cc is the common condition that the current node shares with its parent node. ch is the list of child nodes of the current node.*

The relation graph is defined to capture the relations between different EDFAs. For example, the relation graph in Figure 4.5 captures the relations of EDFAs in Figure 4.4. The first node of Figure 4.5 can be represented as $\{\{can\_monitor(TM, changeoftickinformation)\}, \{(< connect, receive, display >, \{connected(TM, GSYS))\}\}, dfa_1, \varnothing, \{dfa_2\}\}$ according to Definition 18.

We decide whether two predicates are compatible based on a direct comparing on corresponding fields of the predicates, as defined in Definition 2. We also rely on the semantic dictionary to decide whether two words are semantically equal. This has been discussed in Section 3.4.6 and we skip the details here. The necessary condition for the orchestration of two EDFAs directly with succession behavior is defined as follows:

Definition 19 (Link Condition) *Let* dfa_a, $dfa_b \in$ *EDFA be two extended DFA, they can be linked if and only if* $\exists$ $as \in dfa_a.AS$: $\exists t$ *ends in* $as \wedge$ $t.PostC \subset dfa_b.PreC \vee \exists$ $as \in dfa_b.AS$: $\exists t$ *ends in* $as \wedge$ $t.PostC \subset dfa_a.PreC$

Given an EDFA, if we cannot find any other EDFA for the same actor that satisfies the link condition, we raise natural language questions to ask users for suggestions. This case usually implies there is some missing precondition/postcondition.

Algorithm 6 describes the procedure to construct a relation graph for a set of EDFAs. We first construct a single-node graph for each input EDFA and add them to the set of graphs $\hat{G}$ (line 1-3). While the graph set has more than one graphs and the graph set is not stabilized (line 5-15), we try to find a pair of graphs which can be linked based on the link conditions defined in Definition 19. The function Comp(condition, graph) checks whether the given predicates "condition" are compatible with the postconditions of any node in the given "graph". If the two sets of predicates satisfies the conditions in Definition 19, they are compatible. If the precondition is a subset of postconditions and users answer yes to our raised questions, they are considered to be compatible. If such a pair of compatible graphs (line 8, 12) is found, we set one graph as the child of another (line 9, 13) and remove the child graph from the graph set (line 10, 14). As long as the graph set is changed, it is said to be not stabilized. If there are more than one graph in the graph set after it is stabilized, which indicates the preconditions and postconditions of those EDFAs are loosely coupled, we raise questions (line 17) to query whether we can merge those graphs by adding a single

Algorithm 6: Build Relation Graph

Input : a set of extended DFA $\widehat{DFA} = \{D_1 \dots D_n\}$
Output: an usage relation graph g for DFAs in $\widehat{DFA}$

```
1  for D_i in DFA^ do
2      g_i ←construct a single-node graph for D_i
3      Ĝ.add(g_i)                                   ▷ Ĝ is the set of graph nodes.
4  while true do
5      while | Ĝ |> 1&& !stabilized do
6          for (g_i,g_j) in Ĝ do
7              stabilized←true
8              if Comp(g_i.PreC, g_j) then
9                  g_j.ch.Add(g_i, cc)
10                 Ĝ-{g_i}
11                 stabilized←false
12             else if Comp(g_j.PreC, g_i) then
13                 g_i.ch.Add(g_j, cc)
14                 Ĝ-{g_j}
15                 stabilized←false
16     if | Ĝ |> 1 then
17         ret←RaiseQuestion()
18         if ret=false then
19             modify the conditions based on answers from users
20             if find compatible graph pair then
21                 stabilized←false
22                 continue
23             break
24         else
25             g←Merge(∀ g_i ∈ Ĝ)
26             break
27     else
28         break
29 return g
```

root node. If the reply is negative, the user can modify the preconditions/postconditions of the graphs, which can be traced back to the use case documents. If the modifications from the user enable new parent-child relation between those graphs (line 20–22), we continue to link those graphs. Otherwise we merge all the graphs in the graph set $\hat{G}$ to obtain a final relation graph (line 24, 25).

During our study of the use cases, we observed that for most of the use cases, the authors of the use case specifications tend to focus on documenting the action steps and ignore the preconditions and postconditions. Consequently the use cases usually have their preconditions and postconditions partially documented; missing or redundant conditions also appear frequently. Therefore, it is extremely helpful to provide a way for the users to correct/complete the preconditions/postconditions in order to improve the integrity of the use case document.

4.4.4 Orchestrate EDFAs

Given a set of EDFAs and the relation graphs for those EDFAs, we conduct a breadth-first traverse on the relation graph and link the EDFA represented by a graph node with the EDFAs represented by all its child nodes.

Algorithm 7 elaborates the procedure of building an overall EDFA from the set of EDFAs of an actor based on its relation graph. The algorithm starts from the starting node of the relation graph g (line 1). We use a queue Q to aid the breadth first search and the procedure continues processing as long as there are elements in the queue (line 3–24). In each iteration, we pick a node from the queue and check whether it has been processed before and whether it has children. To avoid processing the same node multiple times (e.g., for nodes in a cycle), we mark each visited node and skip the visited nodes during the processing (line 5). We also skip nodes which do not have children nodes since they are leaf nodes and have

Algorithm 7: Build Overall EDFA

Input : g: The relation graph for an actor
$\widehat{DFA}$: the set of EDFAs for the actor
Output: DFA_n: the EDFA composed of EDFAs in $\widehat{DFA}$

```
1  Q.InQueue(g.n_i)
2  DFA_n ← g.n_i.EDFA
3  while Q.size()>0 do
4      curN←Q.DeQueue()
5      if visited(curN) || curN.ch.size()=0 then
6          continue
7      for n in curN.ch do
8          Q.InQueue(n)
9          as←FindFinalState(curN.EDFA, n.PreC)
10         T←FindRelatedTraces(curN.EDFA,n.cc, as)
11         α ←T.PostC∪ n.PreC−n.cc
12         if needSplit then
13             create a new accepting state accept_n
14             DFA_n.as.append(accept_n)
15             ChangeTransitionFunction(T, s, accept_n)
16             AddTransitionFunction(accept_n, n.EDFA.init, α)
17             DFA_n.PostM←curN.PostM∪n.PostM−accept_n.PostC
18         else
19             AddTransitionFunction(as, n.EDFA.init, α)
20             DFA_n.PostM←curN.PostM∪n. PostM−as.PostC
21         DFA_n.S←DFA_n.S∪ n.EDFA.S
22         DFA_n.δ ←DFA_n.δ ∪ n.EDFA.δ
23         DFA_n.Σ ←DFA_n.Σ ∪ n.EDFA.Σ
24         DFA_n.AS←DFA_n.AS ∪ n.EDFA.AS
25 return DFA_n
```

been linked with their parent nodes. We link the current node *curN* with its children nodes (line 7-24) and put all its children nodes into the queue (line 8). For each child node n of the current node *curN*, we first locate the accepting state of the EDFA represented by *curN*. The accepting state is to be linked with its child node n (line 9). Then we find the traces T (recall that the postcondition of a EDFA is stored based on the traces) ending in the accepting state as which should be linked with the child EDFA based on the common conditions n.cc (line 10). If the set of traces T returned in this step is a subset of traces which end in the accepting state as, then we need to split the accepting state as (line 12). For example in Figure 4.4 (b), the accepting state p_7 is split twice (into p_7' and p_7'') when orchestrating with the EDFA in Figure 4.4 (c) and the EDFA in Figure 4.4 (d), respectively. To do so, we first create a new accepting state, add it to the accepting state set of the result EDFA, change all the transitions which ends in the original accepting state as to the newly created accepting state (line 13-15). We then add a new transition from the accepting state to the initial state of the child EDFA (line 16, 19) and adjust the postconditions for each accepting state in the new EDFA. The action of the newly created transition is set in line 11. We add the states, transitions, alphabet and accepting states of the EDFA represented by the child node to the new EDFA (lne 21-24).

The final EDFA is a visualization of the overall dynamic behavior of an actor. It can serve as an initial behavior model for its corresponding actor, which can aid the system design and specification based test case generation [90, 107].

4.5 Evaluation

We have implemented our method in a prototype tool in Java[3]. We adopt ZPar [140], a statistical dependency and phrase structure parser, to analyze syntactic information. To

[3]The source code is available in `http://www.comp.nus.edu.sg/~lius87`

evaluate our approach, we conduct an experiment with a use case requirement specification for a real world stock trading system. This use case document is reasonably well maintained, as required to ensure certain level of rigorousness of such software systems. We do believe it is representative of its kind. It contains more than 150 use cases, which is considerably large. The grammar errors and the inconsistency in writing styles present in the document reveals the reality in industry use cases, which also causes difficulties in parsing those sentences. Nevertheless our method achieved good accuracy in natural language parsing. In this chapter, we focus on the part of active learning and user interaction. We refer readers to the Section 3.5 on the result of accuracy in natural language parsing. In this evaluation, we try to answer the following questions:

1. Is our method helpful in improving the quality of use case specifications?
2. Is the amount of user interactions acceptable to specification engineers?

During the experiment, if a question is raised by the tool, we first screen the question to see whether it has an obvious answer which is missed by the parser or it indeed requires an answer from the author of the system. If it is the latter, the question is directed to the right people for clarification. The experiment results are summarized in Table 4.1. The first column shows the actors in the system. Columns 2-4 are the number of use cases for the corresponding actor, the number of nodes of the generated use case relation graph, and the number of states in the final DFA obtained for each actor. Columns 5-7 show the total number of membership queries, the total number of pattern queries we raised to users during the learning process, and the total number of queries we raised to users during generating of use case relation graphs. Column 8 and 9 show the number of missing scenarios and other problems (e.g., missing/redundant preconditions and postconditions) that we identified (which have been confirmed by stakeholders) during the process.

Table 4.1: Results of the case study

Actor	UC	node	state	Q(L^*)	Q(P)	Q(G)	miss s	problem
Broker	3	3	8	0	0	2	0	0
C Monitor	3	2	6	2	1	2	0	0
Exchanger	3	3	12	5	5	1	0	0
T Monitor	10	5	17	34	23	1	2	1
E Monitor	16	4	15	28	15	2	0	14
Trader	58	27	79	141	79	11	5	14
Server	67	49	187	289	237	9	12	23

Question 1: Improvement on Use Case Quality

There are two commonly happened oversights which lead to missing scenarios. The first one is that different ordering of some actions in a use case can lead to different results, however not all possible orderings are considered. For example, for one certain trading strategy, the ordering of "set timer" and "price order" may result in different pricing, and thus matches of orders. However usually this kind of ordering-sensitive scenarios are not fully specified and the reason, as confirmed with the authors of the use case document, is that they simply did not consider all those situations. Five of the missing scenarios found by our approach belong to this category. The second one is missing of some actions in a trace, such as the dashed lines shown in Figure 4.4. We have found 14 this kind of missing scenarios in the use case document. Those missing scenarios may cause barrier in understanding the system functionality throughout the software development life cycle. The authors of the use case document usually assume certain amount of background knowledge on the readers of the document, which results in those missing scenarios.

Relying only on the preconditions and postconditions to obtain usage relations is inadequate, especially when the use cases are loosely coupled (which is usually the case). But we can still find some cases where the conditions are missing or redundantly stated. Miss-

ing preconditions/postconditions affect the integrity of use case specifications. Redundant preconditions/postconditions, meaning we can infer the remaining preconditions/postconditions from one or a subset of the given preconditions/postconditions, may cause confusions and understanding barriers. The reason is that not all repeated conditions are redundant. For example, in one use case, the preconditions are {*login(trader)*, *create(Server, match)*, *receive(server, order)*} and we can obtain the inference relations *create(Server, match)*→*login(trader)* and *create(Server, match)*→*receive(server, order)* from some existing use cases. In this case, *receive(server, order)* is a redundant precondition for the use case while *login(trader)* is not since *login(trader)* is the precondition for any operation the trader can conduct in the system. Our method find this situation during the generation of relation graphs and raise questions for user confirmation. Among all the 52 use cases which have the stated problems, 50 use cases have missing preconditions, which should be properly addressed, and 2 use cases have redundant preconditions stated.

The questions that our method raise are based on the information we extracted or inferred from the use case document, as well as that provided by users. For some use cases, limited information is provided in the document. For example, one common situation is that the precondition of a use case is not specified and the postconditions of it do not provide enough information for us to infer possible preconditions, which we can suggest to users, based on other related use cases. We notice that the reason for this kind of incompleteness is that the engineers who produce the use cases have certain assumptions on the background knowledge of the readers of the use cases. It is more likely that those use cases describe similar functions of the system and thus have similar preconditions. With this belief in mind, we raise one question which include all the use cases of this kind rather than raising one question for each use case, for the purpose of providing more information to the stakeholders to complete the preconditions of those use cases. This is the reason that we have less questions compared to the problems found during the use case graph generation phase.

Question 2: Involvement of Manual Efforts To answer the second question, we discuss the manual efforts involved in our approach. The major manual effort of our approach is to answer three kinds of queries raised. (1) When the active learning algorithm reports some likely missing scenarios, we need to manually check whether the scenario is a real missing scenario or a false negative. The number of membership queries raised, polynomial to the number of states in the DFA and the length of the counterexample [23], can be very large. To reduce the number of membership queries raised to stakeholders, we proposed filtering techniques to filter the traces returned by L^* before raising it to the users, which dramatically reduce the manual checking efforts. (2) We need to check the patterns mined from the negative counterexamples rejected by users. In our evaluation, we set the support value of the Apriori algorithm to be 20% (of total number of input traces). This is actually the best practice that is from our experience of evaluation. There are two reasons for this result. (a) The traces, i.e., the membership queries that are rejected by users, that serve as input to the Apriori algorithm are not duplicate. (b) Our method mine patterns incrementally, which means in each round of the mining process, we only have a subset of traces which we can mine patterns from. Thus it is hard to get any meaningful patterns with high support values. One problem with the low support value is that it is possible that we will get more questions on the patterns that we mined. However, compared to the membership queries reduced (by adopting the mined patterns), the number of pattern questions raised is quite reasonable. (3) During the DFA orchestration, our method may ask questions when we find potential missing or redundant preconditions/postconditions. The number of this kind of questions raised is far less than the number of the other two kinds of questions.

As we can see from Table 4.1, the average number of all three kinds of questions raised for each use case[4] is around 6, which we consider as a reasonable amount of manual effort. We can notice form Table 4.1, the average number of questions raised for the server use cases is larger than the other use cases. The reason is that the use cases which describe the server functions tend to be more complex, with more branch conditions and longer traces. We also find more missing scenarios/traces in the server use cases. This also caused more questions to be raised since adding a trace causes more membership queries to be raised in the L^* algorithm. Since the DFA learned by our method is prefix-closed, all the prefix traces of a missing scenario would be raised as membership queries. That is one of the reasons why we have more membership queries as compared to the missing scenarios found.

4.6 Chapter Summary

In this chapter, we propose methods to improve use case documents written in natural language interactively with the guidance of users/stakeholders. Our method adopts advanced natural language processing techniques and the active learning algorithms in the domain of machine learning. Our method learns a DFA for each actor/agent in the use case description incrementally with the guidance of users. During the learning process, our method finds potential missing scenarios and preconditions/postconditions of use cases. We conduct evaluations with an industry case study and the results show that our method is able to find missing scenarios and redundant conditions, and aids users to provide modifications efficiently.

[4]Computed by (|Q(L^*)|+| Q(P)|+|Q(G)|)/|UC|

Chapter 5

Model Checking Aided Design Verification

We have shown how to find defects and improve requirement specifications in the previous chapters. As another important activity, system design links the user requirement with the coding phase. Considerable manual efforts are involved in the system design phase, which inevitably introduce defects. Those defects are usually nontrivial and are hard to detect manually, especially for complex systems. In this chapter, we explore an automatic verification technique, i.e., model checking, to aid the verification on design models. Our focus is on UML state machines, which capture the dynamic behaviors of system designs and are more closely related to the actual implementation.

5.1 Motivating Example

In the designing phase of a system, behavior models, such as UML state machines are often used to model the dynamic behavior of components of the system. For example, the state

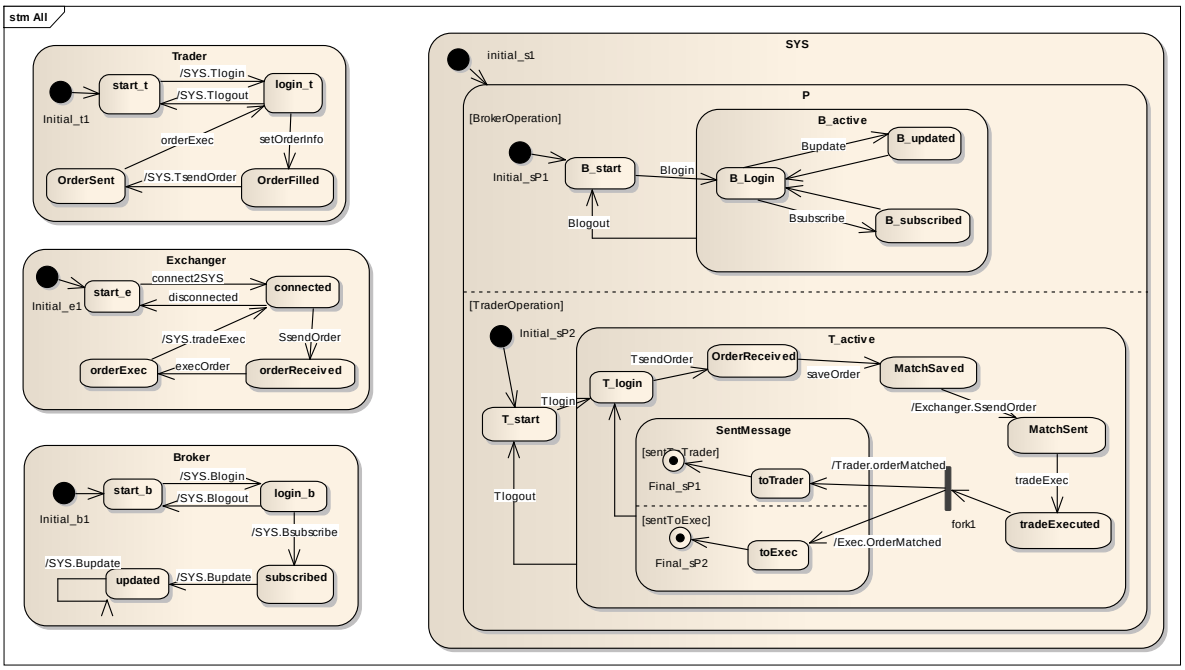


Figure 5.1: State machine for GSYS

machines shown in Figure 5.1 captures the dynamic behaviors of the stock trading system at a fairly high abstraction level[1].

There are certain properties that the system should preserve. For example, a safety property of interest is that the system should be deadlock-free. Another example is related to liveness, i.e., it is required that the broker can conduct the logout action after it had logged in. These properties are not easy to verify manually. Both properties are violated by the state machine in Figure 5.1. However, in the initial design of the stock trading system, the authors did not realize these defects.

LTL Model checking [38] is an automatic verification technique to check a property/constraint specified in Linear Temporal Logic (LTL) against the model specified by Labeled Transition System (LTS). Model Checking is known as a "click button" technique due to

[1]Since the detailed design is highly confidential, we only show an initial design of the system.

its highly automatic verification procedure. It is suitable for checking safety and liveness related constraints. Model Checking has shown its potential in verifying various systems, including security systems [25], reliability systems [58] and Pervasive Computing Systems [94, 95], etc. In this chapter, we explore to conduct model checking on UML state machines to check those safety and liveness related properties in the system design.

5.2 Introduction

UML state machine diagrams are widely used to model the dynamic behavior of an object, which serve as the basis for code development. However, UML specification is documented in natural language, which introduces inconsistencies and ambiguities [52]. The benefit of providing model checking support for UML state machines is two folds. Firstly, it allows early detection of design related problems, such as deadlock, livelock, etc. This is especially important for safety-critical systems. Secondly, it will encourage the consistency usages of UML diagrams throughout the software development process and improve the maintainability. Lacking of tool support is one of the reasons which prevents the usage of UML throughout the software development process (Zeichick [138] found in their survey in 2002 that the reason for one-fourth of the investigated people that do not use UML is because that "their tools do not support UML"). However, the semantics of UML is documented in natural language, which is ambiguous. Fecher et al. [52] discussed 29 unclarities (that is, inconsistencies and ambiguities) in UML 2.0 state machines. But there are still some unclarities (such as the granularity of a transition execution sequence) that are not covered in [52]. Therefore, the first step towards model checking UML state machines is to solve those unclarities and formalize the semantics based on OMG UML specifications [7].

There were some approachs [35, 51, 85, 128] in the literature which provide formal semantics for a subset of UML state machine features. Among all the related works, only a few [51]

have considered UML 2.0 specifications, which has major changes compared to UML 1.x specifications as discussed by [51]. All the related work which considered UML 2.0 specifications cover only a subset of UML state machine features. Moreover, a few approaches (see, e.g., [115, 128]) consider the non-determinism in the presence of orthogonal composite states, which is an important modeling concept in UML state machines. Although extensibility of the syntax structure is important due to the refinement operations on UML state machines, the syntax formats defined in those works does not extend well. A semantics able to support the full set of syntax features, including event pool mechanisms, will help to bring the expressive power of UML state machines to life.

In order to bridge the gaps in the current approaches, we provide a formal operational semantics for the complete set of UML state machine features, which includes formal definition of non-determinism, event pool and the communication mechanisms between different state machines. We also develop a tool which can bring model checking of UML state machine diagrams into practice. The contributions of this work are summarized as follows.

- We provide a formal operational semantics for UML 2.4.1 state machines covering the complete set of UML state machines features. In particular, our syntax structure is extensible to state machine refinement and future changes. Our semantics formalization considers non-determinism as well as synchronous and asynchronous communications between state machines.
- We explicitly discuss the event pool mechanisms and consider deferral events as well as completion events.
- We report new unclarities in UML 2.4.1 state machines specifications.
- We develop a tool USMMC based on the semantics we have defined; it model checks various properties such as deadlock-freeness and linear temporal logic (LTL) properties. We conduct experiments on our tool and results show its effectiveness.

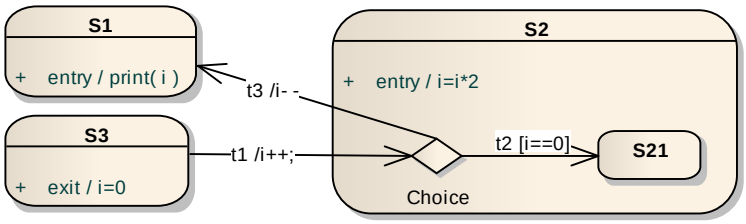


Figure 5.2: Illustration of transition execution sequence

Outline. The rest of this chapter is organized as follows. Section 5.3 provides the preliminaries of UML state machines. Section 5.4 and Section 5.5 define the syntax and semantics for UML state machines, respectively. We introduce the tool USMMC that we developed to conduct model checking on UML state machines in Section 5.6. Section 5.7 provides the evaluation results. Section 5.8 addresses the limitations of our work. We conclude our work in section 5.9.

5.3 Basic Asumptions on UML State Machine Semantics

We briefly sketch below some new unclarities we found in the UML 2.4.1 state machines specification, as well as our assumptions in this chapter.

Transition Execution Sequence. Transitions and compound transitions are used in interleaving in the descriptions of transition execution sequence, which raises confusions. The transition execution ordering is important since different execution orders may lead to different results. For example in Figure5.2, Suppose S_3 is active and transition t_1 is fired. If we define the transition execution sequence based on the compound transition, the behaviour execution sequence is "$i = 0;\ i++;\ i--;\ print(i)$ " and 0 should be printed. If we define the transition execution sequence based on a single transition, the behaviour execution sequence should be "$i = 0;\ i++;\ i = i * 2;\ i--;\ print(i)$" and 1 should be printed. In the first case, the entry behaviour of state *S2* is not executed, which contradicts

Table 5.1: Type notations

Symbol	Type	Symbol	Type	Symbol	Pseudostate type
$\mathcal{K}_{\mathcal{S}}$	active state configuration	$\mathbb{B}$	boolean	DH_{ps}	deep history
$\widetilde{T}$	compound transition	C	constraints	I_{ps}	initial
$\mathcal{K}$	configurations	S_f	final state	C_{ps}	choice
$\langle \widetilde{T} \rangle$	compound transition list	S	state	Jo_{ps}	join
V	vertex	$Trig$	triggers	Ju_{ps}	junction
$\mathcal{K}_{\mathcal{V}}$	active vertex configuration	T	transition	T_{ps}	terminate
CR	connection point reference	E	event	En_{ps}	entry point
SM	state machine	R	region	F_{ps}	fork
B	behaviours	PS	pseudostate	SH_{ps}	shallow history
$\langle B \rangle$	behaviour list	$\mathbb{N}$	natural number	Ex_{ps}	exit point

the semantics of entry behaviours. We define the transition execution sequence based on a transition to keep the semantics consistent with entry behaviours.

Basic Interleave Execution Step. If multiple compound transitions in orthogonal regions are fired by the same event, it is unclear in what granularity should the interleaving execution be conducted: either on transition or on compound transition level. The execution order of the (behaviours associated with the) fired transitions may affect the value of global shared variables. We decide to regard a compound transition as the interleaving execution step, since a compound transition is a semantically complete path.

5.4 Formal Syntax for UML State Machines

In this section, we provide formal syntax definitions for UML state machines features and abstractions of event pools. We define a self-contained model which includes multiple state machines. Table 5.1 lists the basic notations of types defined in this chapter.

Our syntax definition preserves the structure specified by [7], which makes it suitable to support refinement as well as future changes of UML state machines.

Definition 20 (State) *A* state *is a tuple* $s = (\widehat{r}, \widehat{t_{def}}, \alpha_{en}, \alpha_{ex}, \alpha_{do}, \widehat{en}, \widehat{ex}, \widehat{cr}, sm, \widehat{t})$ *where:*

- $\widehat{r} \subset R$ *is the set of regions directly contained in this state,*
- $\widehat{t_{def}} \subset Trig$, $\alpha_{en} \in B$, $\alpha_{ex} \in B$ *and* $\alpha_{do} \in B$ *are the set of deferred events, the entry, exit and do behaviours defined in the state, respectively.*
- $\widehat{en} \subset En_{ps}$ *and* $\widehat{ex} \subset Ex_{ps}$ *are the set of entry point and exit point pseudostates associated with the state.*
- $\widehat{cr} \subset CR$ *is the set of connection point references belonging to the state.* $sm \in SM$ *is the state machine referenced by this state; the two fields are used only when the state is a submachine state.*
- $\widehat{t} \subset T$ *is the set of internal transitions defined in the state.*

There are four kinds of states, viz., simple state (S_s), composite state (S_c), orthogonal composite state (S_o) and submachine state (S_m). In Figure 2.3, the submachine state *Departure* is denoted as $(\varnothing, \varnothing, \epsilon, \epsilon, \epsilon, \varnothing, \varnothing, \{EntryP1,\ ExitP1\}, DepartureSM, \varnothing)$, where ϵ and $\varnothing$ denote the empty element and the empty set, respectively.

Definition 21 (Pseudostate) *A* pseudostate *is a tuple* $ps = (\iota, \widehat{h})$, *where* $\iota \in R \cup SM$ *is the region or state machine in which the pseudostate is defined, and* $\widehat{h} \in S$ *is an optional field which is used to record the last active set of states. This latter field is only used when the pseudostate is a shallow history or deep history pseudostate.*

The last column of Table 5.1 shows the notations of the ten kinds of pseudostates PS.

Definition 22 (Final state) *A* final state *is a special kind of state, which is defined as a tuple* $s_f = (\iota)$ *where* $\iota \in S_o \cup S_c \cup SM$ *is the composite state or state machine which is the direct ancestor of the container of the final state.*

Definition 23 (Connection Point Reference) *A* Connection Point Reference *is defined as a tuple* $(\widehat{en},\ \widehat{ex},\ s)$ *where* $\widehat{en} \subset En_{ps}$ *and* $\widehat{ex} \subset Ex_{ps}$ *are the entry point and exit point pseudostates corresponding to this connection point reference, and* s *is the submachine state in which the connection point reference is defined.*

For example, in Figure 2.3, *EntryP1* is defined as $(\{EntryPoint1\}, \varnothing, DepartureSM)$.

Vertex $V \triangleq S \cup S_f \cup PS \cup CR$ is an abstraction of all nodes.

Definition 24 (Transition) *A* transition *is a tuple* $t = (sv,\ tv,\ \widehat{tg},\ g,\ \alpha\ ,\ \iota,\ \widehat{tc})$ *where:*

- $sv \in V$, $tv \in V$ *are the source and target vertex of the transition, respectively.*
- $\widehat{tg} \subset Trig$, $g \in C$, $\alpha \in B$ *and* $\iota \in R$ *are the set of triggers, the guard, the associated behaviour and the container of the transition, respectively.*
- $\widehat{tc}$ *is a set of tuples of the form* $segt = (ss, \alpha_{st}, \iota_{st})$. *It represents the special situation that a join or fork pseudostate*[2] *connects multiple transitions to form a compound transition. Each tuple represents a segment transition which ends in the join (resp. emanates from the fork) pseudostate.* $ss \in S$ *is the non-fork (resp. non-join) end of the segment transition,* $\alpha_{st} \in B$ *is the behaviour associated with the segment transition.* $\iota_{st} \in R$ *is the container of the segment transition.*

We define the following functions on transitions for clarity sake. Functions $isFork(t)$ and $isJoin(t)$ decide whether transition t is a fork transition and join transition, respectively. For example, in Figure2.3, the join transition *t10* is $(\{Join1\}, \{ExitPoint1\}, \varnothing, \epsilon, \epsilon, RD, \{(SyncExit, \epsilon, RD), (SyncCruise, \epsilon, RD)\})$. We use $t.\widetilde{\alpha}$ to represent all possible action execution sequences of t. Formal definition of $t.\widetilde{\alpha}$ is defined in Appendix A.

[2]We treat exit (resp. entry) point pseudostate the same way with join (resp. fork) pseudostate.

Definition 25 (Region) *A* region *is defined as a tuple* $r \triangleq (\widehat{v}, \widehat{t})$*, where* $\widehat{v} \subset (S \cup PS \cup S_f)$*,* $\widehat{t} \subset T$ *are the set of vertices and transitions directly owned by the region.*

Definition 26 (State Machine) *A* state machine *is defined as* $SM \triangleq (\widehat{r}, \widehat{cp})$*, where* $\widehat{r} \subset R$*,* $\widehat{cp} \subset En_{ps} \cup Ex_{ps}$ *are the set of (directly owned) regions and the set of entry/exit point pseudostates defined for this state machine.*

For example in Figure 2.3, state machine *DepartureSM* is $(\{RD\}, \{EntryPoint1, ExitPoint1\})$.

Definition 27 (Compound Transition) *A* compound transition *is a "semantically complete" path composed of one or multiple transitions connected by pseudostates. The set of compound transition* $\widetilde{T} = \{\tilde{t} \mid \tilde{t} \in ST \wedge \tilde{t}.\widehat{sv} \in S \wedge \tilde{t}.\widehat{tv} \in S\}$ *where* $st \in ST \equiv (len(st) = 1 \wedge seg(st, 0) \in T) \vee \exists\, st_i, st_j \in ST : last(st_i) = first(st_j) \wedge st = st_i \frown st_j$*.*

The operator $\frown$ denotes the operation of connecting transitions in order. Notation $len(\tilde{t})$ denotes the total number of segment transitions the compound transition is composed of. $seg(\tilde{t}, i)$ denotes the ith segment specified by the natural number index i of a given compound transition. We use $first(\tilde{t})$ and $last(\tilde{t})$ to denote the first and last segment of $\tilde{t}$. We define $\tilde{t}.\widehat{sv} = first(\tilde{t}).\widehat{sv}$, $\tilde{t}.\widehat{tv} = last(\tilde{t}).\widehat{tv}$ for convenience sake. For example, in Figure 5.2, the transition sequence $t_1 \frown t_2$ is a compound transition.

Compositional Operators. The operator "; " represents a sequential composition. Interleave operation ($|||$) represents a non-determinism in the execution orders. Interleave with synchronous communications ($|||^C$) is a special case of interleaving: it requires the state machines to synchronize on the specified event in C. Interruption (∇) is used to represent interruption of a do activity by some event occurrence. Parallel composition ($\|$) represents a real concurrency, i.e., execute at the same time.

Definition 28 (System) *A* system *is a set of state machines executing in interleaving (with synchronous communications). sys* $\triangleq |||^{C}_{i\in[1,n]} Sm_i$ *where* $Sm \triangleq (sm, P, GV)$. *In* Sm, sm *denotes the state machine,* P *the event pool associated with* sm, *and* GV *the shared variables of* sm. *And* n *is the number of state machines within the system* sys.

For example, the *RailCar* system in Figure 2.3 is defined by $|||^{C}(Car, Handler)$, where $C = \{departReq, departAck, arriveReq, arriveAck\}$.[3]

Event Pool Abstraction. Change events, signal events, and deferred events are processed differently in UML state machines. We provide for this purpose 3 separate event pools, viz., completion event pool (CP), deferred event pool (DP), and normal event pool (NP). $P \triangleq (CP, DP, NP)$ represents the event pool, and we define two basic operations on P. $Merge(e, EP)$ merges an event e into the corresponding event pool represented by EP, and $Disp(P)$ dispatches an event from P. Since function *Merge* (formally defined in Appendix A) is straightforward, we focus here on Function *Disp*.

Definition 29 *The following function formally defines the event dispatch mechanism.*

$$Disp(P, ks) \triangleq \begin{cases} CP\backslash\{e\};\ CheckDP(P, ks) & \text{if } CP \neq \varnothing \wedge HighestPriority(e, CP) \wedge e \in E \\ DP\backslash\{e\};\ CheckDP(P, ks) & \text{if } CP = \varnothing \wedge DP \neq \varnothing \wedge !isDeferred(e, ks) \wedge e \in E \\ NP\backslash\{e\};\ CheckDP(P, ks) & \text{if } CP = \varnothing \wedge allDefer(DP, ks) \wedge NP \neq \varnothing \wedge e \in E \\ \epsilon & \text{otherwise} \end{cases}$$

$$CheckDP(P, ks) \triangleq DP\backslash E;\ NP \cup E, \quad \text{where } E \triangleq \{e \mid e \in DP \wedge !isDeferred(e, ks)\}.$$

The function guarantees that the precedence order $CP \prec DP \prec NP$ is preserved ($\prec$ denotes the preceding partial order). But the order within each event pool is not specified. The

[3]We treat the state machine (*DepartureSM*) that is referenced by a submachine state (*Departure*) the same way as a composite state.

macro $HighestPriority(e, CP)$ denotes that event e has the highest priority in CP, which preserves the priority order of a nested state over its ancestor states. In the deferred event pool, only events that are not deferred in the current active state configuration ($!isDeferred(e, ks)$) can be dispatched. The macro $allDefer(DP, ks) \Leftrightarrow \forall\, e \in DP, isDeferred(e, ks)$ guarantees the priority of deferred events over normal events. When an event is dispatched, we check all the deferred events defined in the states of the current active state configuration, and remove those events that are not deferred any more from DP to NP; this is accomplished by $CheckDP$.

5.5 Formal Semantics of UML State Machines

This section devotes to a self-contained formal semantics for all UML state machines features. We have adopted the semantic model of Labelled Transition Systems (LTS). The dynamic semantics of a state machine is captured by the execution of RTC steps, which have two kinds of effects, viz., changing active states and executing behaviours. We formally define the two kinds of effects separately. Then the semantics of the RTC step is defined formally. At last, we define the semantics of the system.

5.5.1 Active State Configuration Changes

An active state configuration $\mathcal{K}_\mathcal{S}$ is a set of states which are active at the same time. It describes a stable state status when the previous RTC step finishes. We use Active Vertex Configuration $\mathcal{K}_\mathcal{V}$ (a set of vertices that are active at the same time) to represent the snapshot of a state machine during an RTC execution. For example, in Figure 2.3, {*Operating, Choice2*} is an active vertex configuration. $\mathcal{K}_\mathcal{S}$ and $\mathcal{K}_\mathcal{V}$ are defined in Appendix A.

Next Active State Configuration. $NextK : \mathcal{K}_\mathcal{S} \times \langle \widetilde{T} \rangle \rightarrow \mathcal{K}_\mathcal{S}$ computes the next active

state configuration after executing the compound transition list indicated by $\langle \widetilde{T} \rangle$. Formally: $NextK(ks, (\tilde{t}_1;\ \ldots;\ \tilde{t}_n)) \triangleq NxK(ks_n, \tilde{t}_n)$, where $\forall\, i \in [2, n], ks_i = NxK(ks_{i-1}, \tilde{t}_{i-1}) \wedge ks_1 = ks$. Function $NxK : \mathcal{K}_\mathcal{S} \times \widetilde{T} \to \mathcal{K}_\mathcal{S}$ computes the next active state configuration after executing a compound transition indicated by $\widetilde{T}$. Formally: $NxK(ks, \tilde{t}) \triangleq NxPK(kv_n, seg(\tilde{t}, n))$, where $n = len(\tilde{t})$, $kv_1 = ks$, and $\forall\, i \in [2, n], kv_i = NxPK(kv_{i-1}, seg(\tilde{t}, i-1))$. Function $NxPK : \mathcal{K}_\mathcal{V} \times T \to \mathcal{K}_\mathcal{V}$ computes the next active vertex configuration after executing a transition. Formally: $NxPK(kv, t) \triangleq kv \backslash Leave(kv, t) \cup Enter(t)$. Functions *Leave* and *Enter* represent the set of states left and entered after executing a transition and are defined in Appendix A.

5.5.2 Behavior Execution

Another effect of executing an RTC step is to cause behaviors to be executed. We define the following functions to collect the behavior execution sequence.

Exit Behavior. $ExitBehaviour : \mathcal{K}_\mathcal{V} \times T \to \langle B \rangle$ collects the ordered exit behaviors of states that a given transition leaves in the current vertex configuration. Formally:

$$ExitBehaviour(kv, t) = ExV(kv, MainSource(t), t)$$

$$ExR(kv, r, t) \triangleq \begin{cases} SH(h, v);\ ExV(kv, v, t) & \text{if } r \in R \wedge \exists\, v \in r.\widehat{v} : v \in kv \wedge \\ & \quad v \in S \wedge \exists\, h \in SH_{ps} : h \in r.\widehat{v} \\ DH(h, v);\ ExV(kv, v, t) & \text{if } r \in R \wedge \exists\, v \in r.\widehat{v} : v \in kv \wedge v \in S \\ & \quad \wedge \exists\, h \in DH_{ps} : isAncestor(h.\iota, r) \\ & \quad \wedge\ isAncestor(t.\iota, h.\iota) \\ ExV(kv, v, t) & \text{if } r \in R \wedge \exists\, v \in r.\widehat{v} : v \in kv \\ & \quad \wedge \forall\, s' \in r.\widehat{v}, s' \notin SH_{ps} \\ & \quad \wedge \nexists h \in DH_{ps} : isAncestor(h.\iota, r) \\ & \quad \wedge\ isAncestor(t.\iota, h.\iota) \end{cases}$$

$$ExV(kv, v, t) \triangleq \begin{cases} |||^{C}_{r \in v.\widehat{r}} ExR(kv, r, t);\ exit(v) & \text{if } v \in S_o \vee (v \in S_m \wedge v.\widehat{r} \neq \varnothing) \\ ExR(kv, r, t);\ exit(v) & \text{if } v \in S_c \vee (v \in S_m \wedge v.\widehat{r} \neq \varnothing) \\ exit(v) & \text{if } v \in S_s \\ ExV(kv, cr, t) & \text{if } v \in Ex_{ps} \wedge \\ & \exists\, cr \in CR : v \in cr.\widehat{ex} \\ ExV(kv, v.s, t) & \text{if } v \in CR \\ Agn(v.\widehat{r}, v.sm.\widehat{r});\ ExV(kv, v, t) & \text{if } v \in S_m \wedge v.\widehat{r} = \varnothing \\ \epsilon & \text{otherwise} \end{cases}$$

The exit behaviours of executing a transition are collected recursively starting from the innermost state. We define functions ExV and ExR to recursively collect exit behaviours. All the regions of a composite state should be exited before it. If the region contains a (shallow/deep) history pseudostate, the content of the history pseudostate should be set properly (by functions SH and DH respectively) before exiting the region. Exiting simple states means terminating the do behaviour (if any) and executing the exit behaviour, as defined by $exit(v) = v.\alpha_{do} \nabla v.\alpha_{ex}$. If an exit point pseudostate is encountered, the associated connection point reference is exited, which means the state defining the connection point reference is exited. Exiting a submachine state means exiting all the regions in the state machine it refers to. Function $Agn(v.\widehat{r}, v.sm.\widehat{r})$ assigns the set of regions of a state machine to the the of regions of a submachine state.

Entry Behaviour. $EntryBehaviour : T \rightarrow \langle B \rangle$ collects the ordered entry behaviours of the states a given transition enters. Formally:

$EntryBehaviour(t) = EnV(MainTarget(t), Enter(t))$

$EnR(r, \widehat{V}) \triangleq EnV(s', \widehat{V})$ where $r \in R \wedge s' \in r.\widehat{v} \wedge s' \in \widehat{V}$

$$EnV(v, \widehat{V}) \triangleq \begin{cases} v.\alpha_{en};\ (|||^{C}_{r \in v.\widehat{r}} EnR(r, \widehat{V}) \parallel v.\alpha_{do}) & \text{if } v \in S_o \vee (v \in S_m \wedge v.\widehat{r} \neq \varnothing) \\ v.\alpha_{en};\ (EnR(r, \widehat{V}) \parallel v.\alpha_{do}) & \text{if } v \in S_c \vee (v \in S_m \wedge v.\widehat{r} \neq \varnothing) \\ v.\alpha_{en};\ v.\alpha_{do} & \text{if } v \in S_s \\ GenEvent(v.\iota) & \text{if } v \in S_f \wedge \forall\, r \in v.\iota.\widehat{r}, \\ & \exists\, s' \in r.\widehat{v} : s' \in kv \Rightarrow s' \in S_f \\ Agn(v.\widehat{r}, v.sm.\widehat{r});\ EnV(v, \widehat{V}) & \text{if } v \in S_m \wedge v.\widehat{r} = \varnothing \\ EnV(v.s, \widehat{V}) & \text{if } v \in CR \\ EnV(cr, \widehat{V}) & \text{if } v \in En_{ps} \wedge \exists\, cr \in CR : v \in cr.\widehat{en} \\ \epsilon & \text{otherwise} \end{cases}$$

Entry behaviours are collected in a similar manner to exit behaviours, except that the collect starts from the outermost state. We define functions EnV and EnR to recursively collect the entry behaviours of all the vertices in $\widehat{V}$ in order. States entered by firing transition t are computed by function $Enter(t)$. Starting from the main target state of a transition, all regions of a composite state are entered in interleaving. Entering each state means executing its entry behaviour followed by its do activities ($s.\alpha_{en};\ s.\alpha_{do}$). Do activities of a composite state should be executed in parallel ($\parallel$) with all the behaviours of its containing states. Function $GenEvent(s)$ generates a completion event for state $s.\iota$ and merges the generated event in the completion event pool (CP).

Collect Actions. $CollectAct : \mathcal{K}_{\mathcal{S}} \times \widetilde{T} \rightarrow \langle B \rangle$ collects the ordered sequence of behaviours associated with the execution of the given compound transition. Formally:
$CollectAct(ks, \tilde{t}) \triangleq Act(kv_1, seg(\tilde{t}, 1));\ \ldots;\ Act(kv_i, seg(\tilde{t}, i));\ \ldots;\ Act(kv_n, seg(\tilde{t}, n))$, *and*
$Act(kv, t) \triangleq ExitBehaviour(kv, t);\ t.\widetilde{\alpha};\ EntryBehaviour(t)$ where $n = len(\tilde{t})$, $kv_1 = ks$ and $kv_i = NxPK(kv_{i-1}, seg(\tilde{t}, i-1))$ for $i \in [2, n]$.

5.5.3 The Run to Completion Semantics

The effects of an RTC step execution include both active state changes and behaviour executions which may cause the event pool and global shared variables to be updated. We use the term *configuration* to capture the stable status of a state machine.

Definition 30 *A* configuration *is a tuple* $k = (ks, P, GV)$ *where* ks *is the active state configuration,* P *is the event pool and* GV *is the set of valuation of global variables.*

For example, $(\{Idle\}, (\varnothing, \varnothing, \{setDest\}), \{stopNum = 0, mode = false\})$ is a configuration. The execution of an RTC step can be depicted as moving from one configuration to the next configuration. We provide the following rules to formalize an RTC step. We use the *RailCar* system in Figure 2.3 to illustrate the following RTC step rules.

Wandering Rule. This rule captures the case where a dispatched event e is neither consumed nor delayed. As a result, it is discarded.

$$\frac{e = Disp(P), P' = P \backslash \{e\}, \forall\, s \in ks, e \notin s.\widehat{t_{def}}, Enable((ks, P', GV), e) = \varnothing}{(ks, P, GV) \xrightarrow{e} (ks, P', GV)}$$

Event e is dispatched from event pool ($Disp(P)$), but no transition is triggered by e (i.e., $Enable((ks, P', GV), e) = \varnothing$), and no deferred event in the current configuration matches the event e (i.e., $\forall\, s \in ks, e \notin s.\widehat{t_{def}}$).

Deferral Rule 1. This rule captures the case where a dispatched event is deferred by some states in the current active state configuration, but does not trigger any transitions.

$$\frac{\begin{array}{l} e = Disp(P), P' = P \backslash \{e\}, \exists\, s \in ks : e \in s.\widehat{t_{def}}, Enable((ks, P', GV, e) = \varnothing, \\ P'' = Merge(e, P'.DP) \end{array}}{(ks, P, GV) \xrightarrow{e} (ks, P'', GV)}$$

Since event e is deferred, it should be merged back to the deferred event pool (i.e., $Merge(e, P'.DP)$). So after the RTC execution, only the event pool is changed to P''.

Deferral Rule 2. This rule captures the case where the dispatched event e triggers some transitions and it is also deferred by some states in the current active state configuration. But there exists at least one state, which defines the deferred event, that has higher priority than the source states of the enabled transitions.

$$\frac{\begin{array}{l} e = Disp(P), P' = P\backslash\{e\}, \exists\, s \in ks : e \in s.\widehat{t_{def}}, \widehat{T} = Enable((ks, P', GV, e), \widehat{T} \neq \varnothing, \\ \forall\, \tilde{t} \in \widehat{T} \Rightarrow deferralConflict(\tilde{t}, (ks, P', GV), e), P'' = Merge(e, P'.DP) \end{array}}{(ks, P, GV) \xrightarrow{e} (ks, P'', GV)}$$

$\widehat{T}$ is the set of transitions enabled by the dispatched event e. Event e is also deferred by some states in the current active state configuration and the event deferral has higher priority over transition firing ($\forall\, \tilde{t} \in \widehat{T} \Rightarrow deferralConflict(\tilde{t}, (ks, P', GV), e)$). As a consequence, only the event pool of the state machine changes. For example, ($\{Operating, WaitArriveOK, Watch, WaitEnter\}, (\varnothing, \varnothing, \{opend\}), Env1) \xrightarrow{opend}$ ($\{Operating, WaitArriveOK, Watch, WaitEnter\}, (\varnothing, \{opend\}, \varnothing), Env1$) illustrates the application of this rule, where $Env1$ denotes $\{stopNum = 1, mode = false\}$.

To increase readability, we use the following notations. $\mathcal{A}(\tilde{t}_1, \ldots, \tilde{t}_n) = CollectAct(\tilde{t}_1);$, $\ldots,;$ $CollectAct(\tilde{t}_n)$ denotes the behaviours collection along transitions $\tilde{t}_1, \ldots, \tilde{t}_n$. $Merge(\mathcal{A}(\langle\tilde{t}\rangle), P)$ merges all events generated by actions in $\mathcal{A}(\langle\tilde{t}\rangle)$ into event pool P. Function $UpdateV(\mathcal{A}(\langle\tilde{t}\rangle), GV)$ updates global variables GV by actions in $\mathcal{A}(\langle\tilde{t}\rangle)$.

Progress Rule. This rule captures the case where a set of compound transitions are triggered by a dispatched event e. There is no event deferred, or the fired transitions have higher priority over event deferral.

$$\frac{\begin{array}{l} e = Disp(P), P' = P\backslash\{e\}, \widehat{T} \in Firable((ks, P', GV), e), \mid \widehat{T} \mid = n, \\ \langle\tilde{t}\rangle \in Permutation(\widehat{T}), P'' = MergeA(\mathcal{A}(\langle\tilde{t}\rangle), P'), V' = UpdateV(\mathcal{A}(\langle\tilde{t}\rangle), GV) \end{array}}{(ks, P, GV) \xrightarrow{e} (NextK(ks, \langle\tilde{t}\rangle), P'', GV')}$$

Function $Firable((ks, P', GV), e)$ (defined in Appendix A) returns a set of maximal non-conflicting subset of enabled transitions. The firable set of transitions[4] will be executed in an order specified by $\langle \tilde{t} \rangle$. Function *Permutation* (defined in Appendix A) computes all possible total orders on the set of compound transitions $\widehat{T}$. Behaviors are collected along the transition execution sequence following the permutation order (indicated by $\mathcal{A}(\langle \tilde{t} \rangle)$). Active state configuration is changed as computed by function $NextK(ks, \langle \tilde{t} \rangle)$.

ProgressC Rule. This rule captures the case where choice pseudostates are encountered during an RTC execution. Different from the RTC Progress rule, dynamic evaluation would be conducted at the point where a choice pseudostate is reached.

$$\frac{\begin{array}{l} e = Disp(P), P' = P \backslash \{e\}, \widehat{T} \in Firable((ks, P', GV), e), \mid \widehat{T} \mid = n, \\ \tilde{t}_i^1 \in \widehat{T}, \tilde{t}_i^1.tv \in C_{ps}, \langle \tilde{t} \rangle = (\tilde{t}_1, \ldots \tilde{t}_i^1, \ldots, \tilde{t}_n) \in Permutation(\widehat{T}), \\ GV' = UpdateV(\mathcal{A}(\tilde{t}_1, \ldots, \tilde{t}_i^1)), GV), P'' = MergeA(\mathcal{A}(\tilde{t}_1, \ldots, \tilde{t}_i^1)), P'), \\ \tilde{t}_i^2 \in Firable((\{last(\tilde{t}_i^1).tv\}, P'', GV'), e), P''' = MergeA(\mathcal{A}(\tilde{t}_i^2 \ldots, \tilde{t}_n), P''), \\ GV'' = UpdateV(\mathcal{A}(\tilde{t}_i^2 \ldots, \tilde{t}_n), GV') \end{array}}{(ks, P, GV) \xrightarrow{e} (NextK(ks, \langle \tilde{t} \rangle), P''', GV'')}$$

Compound transition t_i is split by a choice pseudostate into t_i^1 and t_i^2. The second half of t_i is evaluated based on environment GV'. In Figure 2.3, $(\{Operating, WaitArriveOK, Watch, WaitDepart\}, (\varnothing, \varnothing, \{opend\}), Env1) \overset{opend}{\dashrightarrow} (\{Operating, Choice2\}, (\varnothing, \varnothing, \varnothing), Env0) \dashrightarrow (\{Idle\}, (\varnothing, \varnothing, \varnothing), Env0)$[5] illustrates the application of this rule.

5.5.4 System Semantics

A UML state machine models the dynamic behaviour of one object within a system. But state machines representing different components of a system may interact with each other. In order to verify the correctness of the overall system behaviours, we need to capture the

[4] We assume the UML state machines obey well-formedness rules. If more than one non-conflicting sets of transitions are fiable, the choice of which set to execute is non-deterministic.

[5] We use $Env0$ to represent the set $\{stopNum = 0, mode = false\}$. The dashed arrow $\dashrightarrow$ represents an instant stop in a choice pseudostate.

message passing sequences between state machines in the system.

Definition 31 (Semantics of a system) *The* semantics of a system *is defined as a Labelled Transition System (LTS)* $\mathcal{L} \triangleq (\mathbb{S}, \mathcal{S}_{init}, \rightsquigarrow)$. *In this expression,* $\mathbb{S}$ *is the set of states of* $\mathcal{L}$. *Each LTS state is a tuple* $(k_1, \ldots, k_n)$ *where* k_i *is the configuration of the state machine* Sm_i *within the system.* $\mathcal{S}_{init}$ *is the initial state of* $\mathcal{L}$. *And* $\rightsquigarrow \subseteq \mathbb{S} \times \mathbb{S}$ *is the transition relation of* $\mathcal{L}$, *defined below.*

$$\frac{|||^{C}_{i\in[1,n]} Sm_i, k_j \rightarrow k'_j}{(k_1, \ldots, k_j, \ldots, k_n) \rightsquigarrow (k_1, \ldots, k'_j, \ldots, k_n)} \quad [\ LTS1\ ]$$

$$\frac{|||^{C}_{i\in[1,n]} Sm_i, k_j \rightarrow k'_j, e = SendSignal(j,l), Merge(e, EP_l)}{(k_1, \ldots, k_l, \ldots, k_j, \ldots, k_n,) \rightsquigarrow (k_1, , \ldots, k'_l, \ldots, k'_j, \ldots, k_n)} \quad [\ LTS2\ ]$$

$$\frac{|||^{C}_{i\in[1,n]} Sm_i, k_j \rightarrow k'_j, e = Call(j,l), e \in C, k_l \xrightarrow{e} k'_l}{(k_1, \ldots, k_l, \ldots, k_j, \ldots, k_n) \rightsquigarrow (k_1, \ldots, k'_l, \ldots, k'_j, \ldots, k_n)} \quad [\ LTS3\ ]$$

All the state machines in the system are executed non-deterministically. Rule LTS1 captures the normal situation that a single state machine is executed without communicating with other state machines. The notation with prime, i.e., k'_j, represents the new configuration after executing an RTC step. Rule LTS2 defines asynchronous communication, i.e., the executing state machine (Sm_j) sends an asynchronous message ($e = SendSignal(j,l)$) to another state machine (Sm_l). The state machine receiving the message merges the message into its own event pool. Rule LTS3 defines synchronous communication. In this case, the callee state machine (Sm_l) is triggered by the call event ($e = Call(j,l), e \in C$). The caller state machine (Sm_j) cannot finish its RTC step until the callee has finished

execution. For example in Figure 2.3, if state machine *Car* and *Handler* are in configuration $(\{Operating,\ Crusing\}, (\varnothing, \varnothing, \{alert100\}, Env1),\ (\{WaitDepart\}, (\varnothing, \varnothing, \varnothing), \varnothing)$ separately and event *alert100* is dispatched and fires transition t_{12}. The behaviour associated with t_{12} invokes a call event (that is $arriveReq = Call(Car, Handler)$) in *Handler* state machine. The *Handler* state machine consumes the call event and execute an RTC step. After applying rule LTS3, the system is $((\{Operating,\ WaitArriveOK,\ Watch,\ WaitEnter\}, (\varnothing, \varnothing, \varnothing),$ $Env1),\ (\{WaitPlatform\},\ (\varnothing, \varnothing, \varnothing), \varnothing))$.

5.6 USMMC: A Model Checker for UML State Machines

We have implemented the formal semantics presented in Section 5.5 in a tool USMMC [5], which is a module in the PAT [120] framework. The tool supports model checking of safety and liveness properties with different fairness assumptions [120]. We have implemented our tool in a way that all the model checking details are hidden from the users so that users without any model checking background are also able to use our tool. Our tool provides user-friendly graphical interfaces. It takes as input a UML state machine diagram in xmi format, which is compatible with existing UML case tools.

Our tool distinguishes itself from existing UML tools with the following features:

- It is a fully automatic tool for model checking UML state machine diagrams directly and provides user friendly graphical interface.
- It supports model checking of safety and liveness properties with fairness assumptions.
- It supports simulation and verification of multiple UML state machines interactions.
- It reports violations directly in terms of UML state machine execution traces, which are intuitive to follow.

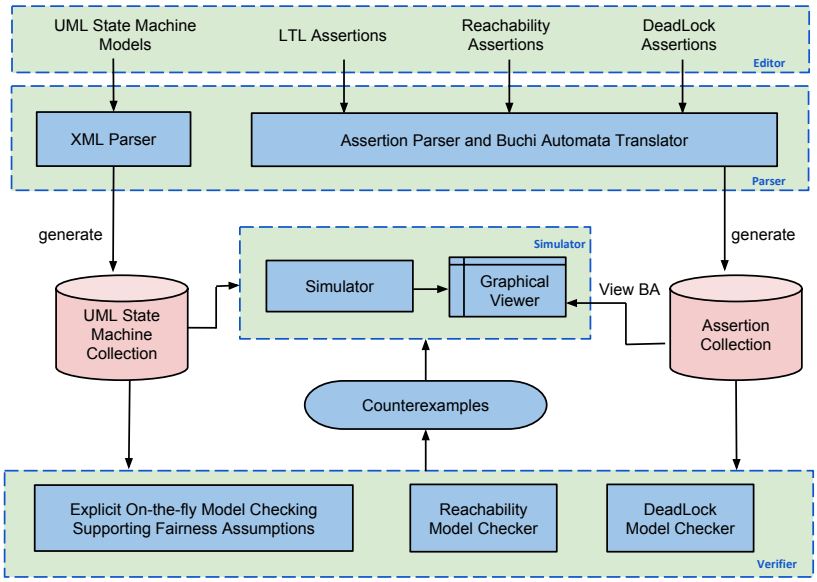


Figure 5.3: The architecture of USMMC

5.6.1 Architecture Design of USMMC

USMMC is a model checker for UML state machines, which consists of four components, i.e., Editor, Parser, Simulator and Verifier. Figure 5.3 shows the architecture design of USMMC. The UML state machines and various assertions can be edited in the text editor component. On clicking the simulation/verification button, the parser is first called to parse the UML state machines and assertions into internal representations; then simulation/model checking components are invoked to perform the simulation/model checking respectively. In the case of an invalid assertion, a conterexample in terms of UML state machines execution trace will be returned. In the following, we are going to introduce each component of USMMC in details.

Editor The editor component provides a text editor which enables editing of both models and properties. Since our tool focuses on providing model checking support for UML state machines instead of graphical modeling, so it takes as input UML state machines in xmi

format, which is exported from existing UML case tools. The editor also supports syntax highlighting and multiple document editing.

Parser In USMMC, we have implemented a parser for UML state machines and a parser for assertions. The xml parser parses UML state machines (in xmi format) into internal representations, i.e., the syntax structure for UML state machines defined in Section 5.4. The formal semantics of UML state machines is implemented to obtain the LTS models from the internal representations. The LTL models will be consumed by simulation and model checking components in later stages. The assertion parser parses LTL, liveness and deadlock free assertions into internal representations. It supports an assertion language which allows LTL formulae constituted with propositions and events, which compliments existing model checkers. Buchi Automata can be generated from negations of LTL assertions.

Simulator USMMC provides a user friendly simulator which is capable of performing automatic random walking or user-guided step-by-step simulations on UML state machines executions. Since our implementation is based on the formal semantics and the LTS steps are consistent with the UML state machine RTC steps, the LTS model generated by our simulator actually directly reflects the execution of UML state machines. Thus the simulator of USMMC provides good tractability to the original UML state machines model. The simulator also provides other functionalities, such as complete state graph generation, trace playback, counterexample visualization.

Verifier The verifier is capable of conducting on-the-fly explicit state model checking on a variety of properties, including deadlockfree checking, reachability checking and refinement checking [129]. Furthermore, it implements dedicated algorithms for model checking LTL properties under a variety of fairness constraints, including global fairness, event/process level weak fairness and event/process level strong fairness [120], which are often required for verification of liveness properties. Counterexamples are returned as UML state machine execution traces (instead of any intermediate formats), which are intuitive to follow.

5.6.2 Implementation Choices for USMMC

There are many "semantic variation points" [52] in UML state machines specifications, which introduce difficulties to the semantic formalization as well as tool implementation. We are going to discuss our choices for implementing some of the important semantic variation points in USMMC in this section.

Event Dispatching Mechanism The concept of event pool is introduced to control the event consumption within a UML state machine and to enable communications between different state machines. It guarantees the stabilization of the state machines within a finite number of steps. But the concrete data structure to implement the event pool and event dispatching order are not specified. Our implementation of event pool follows the operational semantics of UML state machines defined in Section 5.5, where sets are used to represent event pools. Completion and deferred events are properly considered according to the specification [7].

Conflicting Transitions In UML state machines, multiple transitions can be enabled by a single dispatched event. A maximal non-conflicting set of transitions should be decided to execute and the OMG UML state machines documentation [7] has provided two priority rules to deal with conflicts. However not all conflicts are solvable by the provided priority rules. In our implementation, all the enabled maximal non-conflicting transition sets will be enumerated if more than one maximal non-conflicting transition sets are triggered by the same dispatched event and the priority of them cannot be solved.

Communications between State Machines USMMC supports verification and simulation of multiple state machines communication through signal and call events. In our implementation, signal events are processed asynchronously and call events are processed synchronously, i.e., the caller state machine will be blocked until the callee state machine finishes its execution and returns control. Recursive calls are supported in our implemen-

Table 5.2: Evaluation results

Model	Property	USMMC			HUGO			
		Time(s)	State	Transition	TTime(s)	ETime(s)	State	Transition
RailCar	Prop1	0.013	30	34	-	-	-	-
RailCarO	Prop1	0.011	44	54	-	-	-	-
BankATM	Prop2	0.009	25	28	0.231	0.050	578	1,133
TollGate	Prop3	0.110	36	50	0.197	0.505	61,451	256,807
DP2	deadlock	0.005	39	65	0.196	0.111	12,766	42,081
DP3	deadlock	0.039	237	589	0.242	379.009	4,626,838	23,897,077
DP4	deadlock	0.34	1,519	5,079	1.117	8944.754	57,213,708	339,761,530
DP5	deadlock	3.11	9,634	40,366	-	-	-	-
DP6	deadlock	27.87	63,069	324,275	-	-	-	-
DP7	deadlock	232.64	398,101	2,385,361	-	-	-	-

Prop1=$\Box(alert100 \rightarrow \Diamond arriveAck)$, Prop3=$\Box(TurnGreen \rightarrow \Diamond carExit)$,
Prop2=$\Box(retain \rightarrow ((!cardValid \wedge numIncorrect \geq maxNumIncorrect))$

tation.

We developed a model checker and simulator for UML state machines, which is able to check safety and liveness properties and conduct step-wise simulation of UML state machines executions. Our tool is implemented based on a formal operational semantics defined for UML state machines. The experiment results show the effectiveness and efficiency of our tool. USMMC has been implemented as a stand-alone tool in C# with user-friendly graphical interfaces. Starting from 2012, USMMC has come to a stable stage with solid testing and 11 built-in examples. It has been applied to verify many real-time systems ranging from classical concurrent algorithms, such as the dining philosopher problem, to real world problems, such as the railcar system. Our future works include further reducing the state space through techniques such as partial order reduction. Detail information about USMMC (including the video demonstration and the deliverable tool) can be found in our website `www.comp.nus.edu.sg/~lius87`. We list some known issues about our tool and possible solutions as follows.

5.7 Evaluation

In this section, we conduct verification on some commonly used examples with USMMC and HUGO [75]. HUGO is a tool which translates UML state machines into PROMELA models and utilizes Spin to perform model checking. The latest version of HUGO is based on Spin4.3.0, which is out-of-date. HUGO has compatability problems with Spin5.x and Spin6.x, as a result, the conterexample returned by Spin cannot be translated back to UML execution traces and is hard for human to interpret. The examples we use are *RailCarO* [62], *RailCar* [91] (modifies *RailCarO* to manually introduce bugs. Both examples contain transitions emanating/entering orthogonal composite states, which are not supported by HUGO.), *BankATM* [75], dining philosopher[6] and *TollGate* [79]. The experiments are conducted with an Intel Core $i7-2600$, $3.4GHZ$ CPU and $8GB$ memory machine. Our tool works on Windows7, 64-bit operating system and HUGO works with Spin6.2.3 on Ubuntu10.04 LTS operating system.

The verification results are shown in Table 5.2. TTime represents the time used to translate UML state machines models into Promela. ETime represents the time used by Spin to do model checking. Our tool finds the manually injected bugs in *RailCar* system, which is out of the capability of HUGO since HUGO does not support transitions emanate/enter orthogonal composite states, which are presented in the *RailCar* model. The results also show that our tool out performs HUGO both in execution time and memory consumption on all the examples.

The main reason is that the Promela code generated by HUGO has many local transitions, (thus many local variables are introduced to record the status of the intermediate states, which are memory consuming) which introduce overheads. For example, in the generated *TollGate* promela code, 7 steps are conducted to move from an initial pseudostate to its

[6] We model forks and philosophers as separate UML state machines, which execute in parallel.

target state, but it is actually within one RTC step in the UML state machines semantics. Our tool strictly obeys the RTC semantics of UML state machines, where only one step is taken for the above case. The costs introduced by local transitions are exponential in case of non-determinism, such as the dining philosopher example.

The experiment also shows the scalability of USMMC. We check the deadlockfree property (with breadth first search) on dinning philosopher models from $n = 2$ up to $n = 7$[7], and our tool is capable of finding the deadlock within acceptable amount of time. Spin reports out-of-memory error on the models generated by HUGO when $n \geq 4$. The data in Table 5.2 for $n = 4$ is obtained when we set the search depth as the default value, i.e. $1,000,000$. We can see from the result that USMMC can handle large state spaces caused by non-determinism. Reducing further the state space through techniques such as partial-order reduction is the subject of our future work.

5.8 Limitations

We discuss in the following limitations related to our work.

Compatability problem about XMI format Although OMG had released XML Interchange Format (XMI) as the standard exchange format of UML diagrams, different tools adopt different versions of XMI, which causes compatability problems between the models exported by different tools. The models exported by one case tool cannot be properly displayed by the other tools, as is reported by Lundell et al. [96]. This is a known open issue. Since our tool takes the UML state machine models (in xmi format) exported by those tools as input and it is infeasible for us to support all those incompatable formats, we support the output format of Enterprise Architect in the current stage. Providing our own graphical

[7] n is the number of philosophers.

modeling front-end for UML state machines may thoroughly solve the problem and this is subject to our future work.

Structure of the Event Pool Currently, the event pool is implemented as a set in our tool. We are planning to provide more structures, such as queue, bag, and user-defined structures for the event pool implementation in order to meet more modeling requirements.

Action language In the current implementation, we do not provide any specific language for modeling actions and behaviors of UML state machines, just a subset of Object Constraint Language (OCL), arithmetic and boolean calculations are supported. So we are planning to support more complex languages, such as imperative programming languages (C/C#/java), as the description language of events, actions and activities. This will make our tool coincide with existing graphical UML editing tools and is capable of conveying more meticulous system design concerns.

We believe that communications between objects are error-prone and hard to find manually. The experiment results show that our method can find design errors in the presence of both synchronous and asynchronous communications and is scalable.

5.9 Chapter Summary

In this chapter, we provide operational semantics for the complete set of UML state machines features. Our semantics considers non-determinism as well as the communication aspects between UML state machines, which bridge the gap of current approaches. We have implemented a tool, USMMC, for model checking various properties for UML behavioral state machines. Our tool, USMMC, is developed for the purpose of providing fully automatic and direct model checking functionalities, which is efficient, user friendly and achieves a good coverage of UML state machine features. Since the counterexamples are presented in

terms of event execution traces in UML state machines instead of any intermediate formats, USMMC provides good tractability of design flaws. The experiments show that our tool is effective in finding bugs with communications between different state machines.

Chapter 6

Related Work

We discuss the approaches that are related to this thesis in this chapter. In particular, we review the existing work whose end results are related to the overall goal of this thesis. Besides, we also discuss prior works in relation to the technical contribution made by this thesis.

6.1 Finding Defects in Use Cases

There exists various approaches in the literature which address the problem of finding software defects early, i.e., before implementation. In this section, we reviews the approaches that address the problem of finding defects in use case documents (semi) automatically with the aid of natural language processing (NLP) techniques . These approaches extracts either UML models or other formal formats from the natural language descriptions. Then methods are proposed to assess the quality of the documents and to provide hints to potential inconsistencies/incompleteness within the document descriptions.

Gervasi and Zowghi [56] proposed to uncover inconsistencies in natural language use case

descriptions with formal reasoning techniques. Propositional logic formulas are adopted to represent facts, hypotheses and constrains, which are extracted from natural language use case descriptions. Then inconsistencies are checked by reasoning on the propositional formulas. This work relies on a domain-specific natural language parser CICO [55], which requires the MAS (Model, Action and Substitution) parsing rules to provide domain specific patterns. This work aims at extracting fine-grained rules from use case documents, which assumes the writing style of the sentences to be consistent with the provided MAS rules. However, providing MAS rules for large documents, such as the stock trading system used in our evaluation, would be infeasible due to the variety of sentence patterns. Our work uses a dependency parser, which is trained with more than 200 thousand sentences based on statistical machine learning techniques, thus is more robust in handling sentences with different patterns and writing styles. Sinha et al. [117] proposed an analysis engine for use cases. Shallow parsing techniques are used to analyze natural language sentences. Domain-specific knowledge is required for annotating concepts and context information. The main focus of the work is the accuracy of extracting actions in the form of first order logic. Following this work, Sinha et al. [118] developed a prototype tool called Text2Test, which aims at assisting writing correct use cases. The evaluation showed that Text2Test contributes to writing more compliant and complete use case documents. The works [118, 117] extract subject, object based on patterns defined on already identified word tokens and POS tags. The patterns are thus document dependent. We explore to use dependency parsing technique, which provides richer syntactic information and enables adjusting rules without any domain specific information. Thus our approach is more adaptive.

Xiao et al. [134] extracted Access Control Policies (ACP) from software descriptions and checked the validity of the policies against the use case steps within the same document. Semantic patterns for ACP sentences are provided based on the manually defined verb phrases and POS tags obtained from shallow parsing. ACP sentences indicate restrictions

on resource assessment and thus usually have specific key verbs, such as "allowed", "can read", which indicate the actions. However, our work focuses on extracting action tuples from use case sentences of general purposes, i.e., actions can be arbitrary verbs. It is thus infeasible for us to identify actions based on pre-defined key verbs, especially for large documents (e.g., the stock trading system used in our evaluation). Therefore, we provide more general rules (without any document specific key words) based on natural language grammar to process all possible cases.

6.2 Learning Behavior Models from Scenarios

Another kind of approaches that is related to this thesis is those which extract or learn behavior models from scenarios captured by use cases or by Message Sequence Chart (MSC).

6.2.1 Learning Behavior Models from Scenarios Captured by Use Cases

Kof [77] proposed to generate MSC from natural language use case descriptions. The sentences can be processed are restricted to the simple structure, i.e., subject + verb + object. Passive sentences, subordinate sentences and sentences containing multiple actions (verbs) are all considered as error cases. Then a method was proposed to identify missing objects and actions by analyzing consecutive sentences. Another work [78] by Kof aimed at extracting automata from natural language use case descriptions. The method contains three phases, i.e., identify states, assign categories to segments of sentences, generate transitions. Both approaches identify useful information based on Part of Speech (POS) tagging and several document-specific heuristics are used in the process, which make it hard to generalize the approach to an arbitrary use case document. Both approaches are only evaluated on three use cases. Raji and Dhaussy [111] proposed a framework which automatically generate UML activity diagrams from extended use case descriptions. They first generate

an activity diagram for each use case and then synthesis those diagrams based on actors. The main scenarios of the use case that can be processed are described in structured natural language [102]. Yue et al. [137] proposed a method to automatically generate activity diagrams from Use Case Models (UCM, a set of use cases). They first manually re-write the textual use case models in restricted use case modeling (RUCM [136]) format, which composes of a use case template and 26 well-defined restriction rules. Then the UCM is transformed into an instance of Use Case Meta (UCMeta) model, which is later transformed into activity diagrams. In another work, Yue et al. [135] proposed to automatically generate a system level UML state machines from the use case models for the purpose of model-based test case generation. Both approaches focus on generation of models from the UCMeta model. POS tags and grammatical dependence relations are only used to identify control flow information, i.e., relations between sentences. Fine-grained information, such as actions and message passing information are not considered. Moreover, manual efforts are required to rewrite use cases in natural language into the RUCM format. Gutiérrez [59] proposed a way to generate an activity diagram for a use case scenario described in XML format. A metamodel for functional requirement and a subset of activity metamodel are provided. The transformation rules from functional requirement memtamodel to the activity diagram metamodel are described with QVT-Relationals [17]. Manual efforts are required to rewrite a natural language use case description into XML format. Most existing works [59, 102, 137] which generate UML activity diagrams do not process real natural language, manual rewriting from natural language to some structured formats are required. Moreover, these approaches do not analyze natural language sentences. Therefore the actions of action nodes in the activity diagrams are represented by raw natural language sentences from the use cases. However our focus is to identify defects from the use cases based on activity diagrams. Therefore we explore more advanced NLP techniques to obtain fine-grained information, i.e., action tuples and guard conditions for each sentence in use case flows. Our method also has better generality as compared to approaches [77, 78] which

rely on heuristics and template matching on POS tags.

6.2.2 Learning Behavior Models from Scenarios Captured by MSC

Whittle et al. [132] proposed an approach to generate UML statechart from MSC. The MSCs are explicitly annotated with state vectors which contain the valuations of state variables. This information is crucial to their approach and must be provided by users. In [131], Whittle et al. proposed to map a use case charts [130] to a hierarchical state machines. A set of mapping rules are defined from the notation of each level of the use case charts to state machine features. One potential problems with this approach is that the state machine may have too many levels (at least three levels and easily exceeds to four or more levels), which affects the readability. Our work complements with this work in the sense that the use case relation graph generated by our approach can be used to construct the hierarchical structures which serve as input to this approach. Uchitel et al. [125] proposed to synthesis behavioral models from MSC. Similar to [132], this approach requires annotations, i.e., labeling states and providing continuation relations, on the MSCs. In [124], Uchitel et al. proposed to synthesis behavior models represented by Model Transition Systems (MTS) [83] from both safety properties, which specify the upper bound of system behaviors, and scenarios, which describe the lower bound of the system behaviors. MTS can properly capture the lower and upper bound of system behaviors simultaneously, which provide guidance for requirement elicitation. Mäkinen et al. [97] adopted the L^* [23] algorithm to learn statecharts from MSC. However the desired language is expressed as a set of traces, which is insufficient to express loops. Moreover, their approach does not consider to reduce the number of membership queries. Our approach capture the desired language with an automaton, which naturally captures loops. We also propose filtering techniques to safely reduce the number of membership queries raised to users.

The above approaches all take scenarios captured by MSC as input. However, MSC is a for-

mal structure and is not easy to obtain at first. Usually strong knowledge and experience on UML modeling are required to construct MSC from raw natural language descriptions, which is the initial form of scenarios. Moreover, it is hard for stakeholders to get involved with such a formal structure, which causes difficulties of specification validation. Our approach works directly on scenarios captured by natural language, which facilitate the involvement of stakeholders. Another observation is that the above approaches assume the scenarios to be synthesized start with the same preconditions. But this is usually not true in practice. For example, the user login use case happens before any other user operations, and thus is usually the preconditions of the other use cases. In our approach, we consider the relations between use cases based on their precondition and postcondition relations.

Damas et al. [41] proposed to synthesize LTS from MSCs. They synthesis a global LTS and then project it into local LTS based on different agents. The method modified an existing learning algorithm RPNI [108] to add interactions with users, thus help with scenario elicitation. The learning algorithm does not conduct candidate query and thus suffers from over-generalization problem. To overcome that problem, Damas et al. [42] proposed to inject goals in the form of fluent-based assertions into the synthesize process. Sharing the similar idea with the work by Uchitel et al. [124], the goals/constraints extracted from the domain knowledge help to control the scale of the synthesized model and reduce the number of membership queries. Uchitel et al. [126] proposed an approach which took scenarios in MSC and relations between scenarios described in hMSC as input, then behavior models are synthesized and are used to find implied scenarios.

Rather than generating models/prototype implementations, which is the main purpose of the above revealed approaches, our work aims at improving the quality, especially completeness related aspects, of use case scenarios captured in natural language. We value the involvement of stakeholders, which is critical to the completeness of use case specifications.

6.3 Model Checking on UML State Machines

In this section, we discuss works which provide model checking for UML state machines. The first step towards model checking a UML state machine [7] is to provide formal semantics for it since UML state machine is a semi-formal language with informally specified semantics in natural language. Based on how the semantics are provided, the existing approaches can be categorized into two categories i.e., those translate UML state machines into some existing languages which have formal semantics and those directly define the operational semantics. In this section, we discuss both kinds of approaches.

6.3.1 Translation based approaches

A popular approach to formalize UML state machines is to provide a translation to some existing formal languages (such as Petri Nets [71] or Abstract State Machines (ASM) [48]), or to the specification languages of model checkers (such as Spin [13], UPPAAL [16], SMV [37] and PAT [120]). Those formal languages have their own operational semantics. This kind of approaches can be regarded as an indirect way of providing formalization for UML state machines, i.e., the translation based approaches. We categorize these approaches based on the target formal languages they adopt, viz., abstract state machines, Petri Nets, and translation to the specification languages of model checkers.

6.3.1.1 Translation into Abstract State Machines

Abstract State Machines (ASMs) [48] can offer a general notion of state (which abstract away from graphical form) in the form of "structures of arbitrary data and operations which can be tailored to any desired level of abstraction." UML state machine configuration changes are represented by transition rules, which consists of conditions and update functions. The

notion of multi-agent ASMs can naturally reflect the interaction between objects. In this kind of approaches, model checking of UML state machines relies on the theoretical and tool support for model checking abstract state machines that are provided by Spielmann [119], Castillo and Winter [44] and recently Beckers *et al.* [28].

Börger *et al.* [33, 34, 35] are among the pioneers in formalizing UML state machines into ASMs. Their work [33] adopted tuples as the syntax model, which captures the attributes and associations of a construct (e.g., states, transitions). This approach covers most UML state machines features, including deferred events, completion events and internal activities associated with states. But pseudostates such as fork, join, junction, choice and terminate are not considered. Another work [34] extended the work in [33] to support transitions from and to orthogonal composite states in the context of event deferral. In 2004, Börger *et al.* [35] provided some further discussions about the ambiguities in the OMG documentation of UML state machines [6] (version 1.4) and their solutions. The work by Eörger *et al.* [33, 34, 35] cover a large set of features and the formalization is easy to follow. However no automatic translating tool has been developed based on their work so far. Compton *et al.* proposed an approach [40] which translates UML state machines into extended ASM. Extended ASM extends ASMs to represent inter-level transitions with multiple transitions which do not cross any boundary of states. This extension makes it easier to deal with interruptions. It also makes the formalization procedure more structured and layered (since inter-level transitions break the hierarchical structure of UML state machines and the decomposition of inter-level transitions into multiple transitions preserves such a hierarchical structure). The work shares a similar idea with [33, 34] in the rest of the translation procedure. Jürjens [73] provided a work, which focused on the communication aspects between UML state machines. The work explicitly models the message (with parameters) passing between state machines as well as the event queue. Jin *et al.* [72] provided an approach which syntactically defined UML statecharts as attributed graphs which are described using

the Graph Type Definition Language (GTDL). The semantic model is an Object Mapping Automaton (OMA [70]), which is a variant of ASM. However, this approach supports a limited subset of UML statecharts [1] features, concurrent composite states as well as choice vertex are not considered.

To summarize, approaches translating UML state machines into ASMs tend to support more features such as orthogonal composite states, completion/defer events, fork/join/history/choice pseudostates and inter-level transitions (than other kinds of transition based approaches). The reason may be that ASMs are more flexible in terms of syntax format as well as update rules and are more suitable to express the non-structured feature of UML state machines.

6.3.1.2 Translation into Petri Net

Petri Net [71] is a mathematical modeling language with formal semantics. They are always used in modeling distributed systems where concurrency presents. Several approaches [22, 26, 36, 109] in the literature translate UML state machines into Petri Net.

Baresi *et al.* [26] proposed an approach to formalize UML diagrams, including class diagrams, state machine diagrams and interaction diagrams, with high-level Petri Nets. However, the customization rules for each diagram are not formally defined, they are only illustrated with the Hurried Philosopher Problem [2]. Choppy *et al.* [36] proposed to translate UML state machines to Hierarchical Colored Petri net (HCPN). They provide a detailed pseudo-algorithm for the formalization procedure. States in UML state machines are mapped into Petri Net places. Transitions are mapped to arcs in Petri Net and corresponding triggering

[1] We use UML state machines and UML statecharts in interleave in order to respect the notations used in the surveyed works.

[2] The hurried philosopher problem extends the original dinning philosopher problem by allowing new philosophers to be temporarily invited at the table

events are properly labeled. Though the mapping from UML state machines to HCPN is clearly expressed, a very limited subset of UML state machines features (such as simple state, composite state, transitions, triggering event and entry/exit actions) are supported. How to type the events, and how to deal with concurrency invocations of a concurrent composite state are not discussed. André *et al.* [22] proposed an approach which considered a larger subset of UML state machine features (including state hierarchy, internal/external transitions, entry/exit/do activities, history pseudostates, etc.) compared to the previous work [36]. However, concurrency is not considered. Therefore fork and join pseudostates as well as synchronous communication between state machines cannot be expressed. Petri Net is used in modeling work flow in industry. However its notations are always hard for non-experts to understand. With automatic translators from UML state machines to Petri Net, we can benefit from the rigorous verification power of existing Petri Net verification tools [127]. However, approaches translating UML state machines to Petri Net usually cover a small subset of UML state machine features.

6.3.1.3 Translation into the Specification Language of Model Checkers

Another kind of approaches translate UML state machines into the specification language of some model checkers (e.g., Spin, SMV, UPPAAL and PAT). Bhaduri *et al.* [30] provided a survey on this kind of approaches. But it focuses on many variants of Harel statechart [61, 63, 64], such as Requirement State Machine Language (RSML) [87] and Unified Modeling Language (UML). We rather focus specifically on UML state machines, which is the object-oriented variant of Harel statecharts. In this section, we discuss this kind of approaches. We categorize the approaches based on the specification languages of model checkers they adopted.

Approaches Based on Spin Latella *et al.* are pioneers who contributed to the formal verification of UML state machines. They proposed a translation [84] from UML state machines to Process/Protocol Meta Language PROMELA [11], the specification language of the Spin model checker. The translation function takes a hierarchical automaton as input and generates PROMELA code as output. This approach uses STEP PROMELA process to simulate a run to completion step in UML state machines. The translation process is structured since it is based on the pre-defined formal semantics of EHA [85]. Schönborn *et al.* [114] provided a method to model checking UML state machines as well as collaborations with the other UML diagrams. They compile UML state machines into a PROMELA model and collaborations into a set of Büchi automatas. Then the Spin model checker was invoked to verify the Büchi automata against the model. Each state in the state machine is mapped to an individual PROMELA process. The event queue is modeled as buffered channels and communication among processes are modeled via unbuffered channels (synchronized). In this way, the consistency of collaboration diagram with the state machine diagrams can be checked. Jussila *et al.* [74] provided an approach to translate UML state machines into PROMELA. The work considers communications between objects. It provided an action language, a subset of the Jumbala [46] action language, that is used to specify guard constraints and behaviors. The authors implemented a tool called PROCO [10], that takes a UML model in the form of XMI [8] formats and outputs a PROMELA model.

Approaches Based on SMV and its Variants Kwon [80] first provided a translation from UML statecharts to the SMV [100] model checker by rule-rewriting systems. Compton *et al.* [40] provided another approach which used SMV as the back end model checker to automatically verify UML state machines. The work first translates UML state machines into ASMs. Then model checking (with SMV model checker) is conducted relying on a translation tool from ASMs to SMV [44]. Lam and Padget [81] proposed a symbolic encoding

of UML statecharts. The approach invokes NuSMV [37] to perform the model checking. Beato *et al.* [27] also provided a translation from UML diagrams to the input language of SMV model checker. This work focuses on the collaborations of different UML diagrams such as class diagrams, state machine diagrams and activity diagrams. However, this paper does not describe the detailed translation rules, it only illustrates the translation procedure with an ATM machine example. Dubrovin and Junttila [47] provided a symbolic encoding for UML state machines. Then they provided a translation from the encoding to the input language of the NuSMV model checker. The detailed translation steps are not discussed in the paper. An implementation (SMUML [15]) has been provided, and some experiment results are reported.

Approaches Based on Other Model Checkers Gnesi and Latella *et al.* [57] proposed a translation from a hierarchical automaton into a labeled transition system (LTS). The translation is based on the formalization of UML state machines in their early work [85]. Traoré [122] and Aredo [24] proposed to translate UML state mcahines into PVS (Prototype Verification System) [12], which is a specification language integrated with verification tools capable of doing theorem proving, well-formedness checking and model checking. Knapp *et al.* [76] presented an approach to translate UML state machines into timed automata which is used by the UPPAAL [16] Model Checker. Event queue and UML state machine are separately modeled by timed automata and the communication is modeled with a channel. This approach is implemented in a prototype tool named HUGO/RT [75], which can verify consistency of collaboration diagrams with the corresponding set of timed UML state machines. Ng and Butler [103, 104] proposed to translate UML state machines into CSP [65] and utilize the FDR [3] model checker to conduct model checking. Due to the differences between CSP and UML state machines, some features of UML state machines, such as the priority mechanisms, cannot be modeled. Hansen *et al.* [60] used another model checker, mCRL2 [19], to perform model checking tasks. Their work translates Executable UML (xUML) [101] into

mCRL2 specifications. Zhang and Liu [139] provided an approach which translated UML state machines into CSP#, an extension of the CSP language, which is as the specification language of the PAT [120] model checker. An implementation of the translator was done and experiment results of the verification of UML state machines with PAT were presented.

6.3.1.4 Tool Supports for Model Checking UML State Machines

There were some tools developed for model checking UML diagrams. vUML [89] is one of the early tools which translates UML state machines into PROMELA models. It supports checking of deadlock, livelock, and reachability properties. However, explicit annotating of states with stereotypes and constraints are required to express these properties. Users must understand the translated PROMELA model in order to provide LTL properties. HUGO [75] is a tool which aims at verifying the consistencies of UML state machines with properties specified by collaboration diagrams or sequence diagrams. It translates UML state machines into PROMELA. It supports model checking of deadlock properties and LTL properties. TABU [27] translates UML state machines into the specification language of SMV [100] model checker. Different from vUML and HUGO, it can support verification the of LTL properties by providing property patterns, which guides the writing of properties. Another tool [116] is based on the Cadence SMV Model Checker [1]. It is capable of checking both well-formed rules as well as liveness and safety properties, of UML state machines. JACK [57] is an integrated environment based on the usage of process algebras, automata and temporal logic. It supports many phases of system development process by integrating different editing and verification tools. The AMC component inside JACK is able to conduct model checking UML statecharts against ACTL [105] properties. But the components of JACK use FC2 as exchange format, which is not widely supported by the state-of-practice tools. Among all the tools discussed here, only HUGO is currently

available. All of them, except JACK, conduct a translation-based approach, which suffers from efficiency and tractability problems. JACK, though directly implements the semantics, is not fully automatic and is unavailable now.

6.3.1.5 Summary

The translation approaches aim at utilizing the automatic verification ability of different model checkers. Therefore the advantage of these approaches is that most of them provide the implementations of the proposed transition rules. However, we notice that translation-based approaches suffer from the following weaknesses:

1. Due to the semantic gaps, it may be hard to translate some features of UML state machines, introducing sometimes additional but undesired behaviors. For example in [139], extra events have to be added to each process so as to synchronize the exiting of multiple regions of an orthogonal composite state.

2. For the verification, translation approaches heavily depend on the tool support of the target formal languages. Furthermore, the additional behaviors introduced during the translation may significantly slow down the verification.

3. Lastly, when a counterexample is found by the verification tool, it is hard to map it to the original state machine execution, especially when state space reduction techniques are used.

Following these remarks, we believe that a direct implementation based on an operational semantics will provide better solutions for model checking UML state machines in terms of efficiency and tractability.

6.3.2 Operational Semantics for UML State Machines

Unlike translation based approaches, there are another kind of approaches which directly provide operational semantics in inference rules. These approaches are of general purpose and various verification techniques can be conducted based on the operational semantics. The benefit of this kind of approaches are (1) they do not rely on the target specification languages, thus no redundancies are introduced. (2) the semantic steps defined in the operational semantics coincide with the UML state machines semantic step, i.e., the Run To Completion (RTC) step. Moreover, approaches in this category usually adopt Labeled Transition System (LTS) as the semantic model, which coincides with explicit state model checking techniques.

Latella et al. [85] are among the pioneers who begin to focus on formalizing UML statechart semantics. The semantic model their formalization adopted was Kripke structure. They use a slightly modified variant of Extended Hierarchical Automata (EHA) as an intermediate model and map the UML-statecharts into an EHA. The hierarchical structure of UML statecharts and EHA make the translation structured and straightforward. Then they define the operational semantics for EHA in the domain of Kripke structure. This approach covers a quite restricted subset of UML state machine structures, no pseudostates (exclusive of initial pseudostate) are considered. A problem aroused by using EHA as intermediate model is that transitions from a state to its container composite state or from a composite state to its substate cannot be represented properly. Hierarchical Automata requires a strict hierarchical structure. The existence of inter-level (a transition which crosses multiple states) transitions and local transitions break the hierarchical structure. EHA extends the Hierarchical Automata to handle inter-level transitions by assigning the inter-level transition to the outermost Sequence Automata. Since states in different hierarchies cannot appear in a single Sequence Automata, it is hard (actually not supported in [85]) to express transitions which have their source and target states in different hierarchies. The work [45] extended

Hierarchical Automata to support more features such as entry/exit actions, parameters in actions and provided a formal semantics for a subset of UML statecharts based on the EHA. It considered the non-determinism caused by multiple concurrent state machines, which was not captured by previous approaches [85].

The work [128] formalized a subset of UML statecharts based partially on the work proposed in [85]. However it supports some more features such as history mechanisms, entry and exit actions compared to [85]. The syntax used in this chapter is called a term (Basic term, Or-term and And-term), which contains basic information about a state such as an unique ID, entry and exit actions, and sub-terms (for Or-term and And-term) which contains the hierarchical information of a UML state machine. Inter-level transitions are captured by explicitly specifying source restrictions and target determinations in an Or-term (this notation follows the idea of Latella et al. [85]). The dynamic behaviors of UML statecharts are represented by Configurations. A configuration captures the complete current status of a given UML statecharts term, i.e., the hierarchical structure is considered, meaning all currently active substates within the given term are computed. The semantic model is Labeled Transition System (LTS). Kwon proposed an approach [80] which utilized Kripke Structure as the semantic domain and aimed at model checking UML statecharts. Similar to the work in [128], this work utilized term as the syntax model of UML statecharts. This work [80] adopted the conditional rewrite rules to represent the transition relations in UML statecharts. One limitation of this approach is that only a few features are considered and adding the remaining features is not trivial.

Lilius and Paltor [88] provided an abstract syntax and semantics for a subset of UML state machines. The work adopted terms as syntax model. It considered most features of UML state machines. This work did not define a clear semantic model. It formalized the Run to Completion (RTC) step semantics into an algorithm. However, the algorithm is highly abstract and many concepts such as history pseudostates and completion events are described

informally. Eshuis and Wieringa [49] also utilized LTS as semantic model to provide an operational semantics for UML statecharts. This work focuses more on the communication and the timing aspect of UML statecharts. The approach also defines an action language, which includes assignment, object creation/destruction, sequence operations, signal sending operation and time expressions. Damm *et al.* [43] provided a formal semantics for a kernel set of UML, including static and dynamic aspects of the UML models. The formalization contains two steps. Firstly, real-time UML (rtUML) is represented in terms of the kernel subset of real-time UML (krtUML). Then formal semantics of krtUML is provided. This approach provided a self-defined action language, which supports object creation/destruction, assignment and operation calls. This work provides a good solution for communications between different objects as well as event dispatching and handling. Fecher and Schönbor [51] used core state machine, which is a subset of UML state machines, as the semantic domain for UML state machines. This work [51] firstly formalized both syntax and semantics of the core state machine. Then it provided five steps to transform a UML state machine into a core state machine. The transformation steps from a UML state machine to a core state machine are provided informally. Moreover, the translation is very complex since a lot of auxiliary vertices, such as enter/exit vertices, need to be added. Jens et al. [115] provided a very comprehensive discussion about UML 2.0 behavioral state machines, including discussions about detailed semantics of each feature and the ambiguity statements. This approach covers most features of UML2.0 state machines, except for junction and choice vertices, which are considered as syntax sugar. However, choice vertex cannot be easily represented with existing constructs since it represents a dynamic decision point. Termination pseudostates and completion events are not considered. The syntax model of UML state machine is tuple, which captures the components of each construct. There is no clear semantic domain and the semantics are provided by functions that define the RTC steps.

6.3.3 Summary

We reviewed the works which are related to model checking on UML state machines. Existing approaches are categorized into two kinds, i.e., translation based approaches and approaches which provide formal semantics for UML state machines. Translation based approaches usually suffer from the weakness of efficiency and tractability problems. Those approaches which provide formal semantics for UML state machines only consider a subset of UML state machine features. The comparisons of the supported features of the surveyed approaches are presented in Appendix B.

6.4 Chapter Summary

We discussed the existing approaches that are related to the focus of this thesis, i.e., finding defects in software requirement specifications and design models. We briefly reviewed those existing approaches and discussed our improvements on state-of-the-art.

To summarize, our approaches contribute mainly on two aspects. For requirement specifications, we improved the accuracy and adaptability of information extraction by adopting advanced natural language parsing techniques and proposing grammar-driven rules. We also contribute in finding more kinds of defects, especially incompleteness related defects. We value the involvement of stakeholders, which is a critical factor in the success of a project. For system design, we provide tool-supported model checking on UML state machines. Our approach considers the full set of syntax in the latest version of UML state machines specification, with all the complex features that were not considered by existing approaches.

Chapter 7

Conclusion and Future Work

In this chapter, we first summarize the contributions of this thesis in Section 7.1. Then we discuss the possible future directions in Section 7.2.

7.1 Conclusion

This thesis contributes in providing techniques to improve the quality of software requirement specification and system design. To be specific, this thesis has the following contributions.

- We proposed methods to detect defects in natural language use case descriptions. Compared to existing approaches [56, 134] which are based on document-specific template matching, our work is more adaptable since we propose rules based on general English grammars.
- In addition to inconsistency related defects, we also consider integrity related defects, such as missing alternative flows and preconditions/postconditions, which are not

considered in existing works. We evaluated our method with a real system requirement specification document and the results show that our method achieves good accuracy in finding those defects. The found defects are confirmed by the developers to be real defects.

- We proposed methods which adopt an active learning technique to find defects which involve multiple use cases, e.g., find missing scenarios and missing preconditions/post-conditions, through interactions with users. Compared to existing methods [42, 97] which require MSC as input, our method works directly on use cases written in natural language and has no assumption on users' background knowledge, thus increases the involvement of stakeholders, which is critical to improving the quality of requirement specifications.

- We defined a formal operational semantics [91] for the latest version (v2.4.1) of UML state machines. Our semantics cover all the non-time features of UML state machines and respect the UML metamodel. Compared to existing works [51, 85, 128, 45] which only cover a small subset of features of UML state machines, our work is more solid in terms of feature coverage.

- Based on the operational semantics, we developed a domain-specific model checker USMMC [92]. Our model checker supports simulation of UML state machine executions and explicit model checking of LTL properties. We evaluate our tool with 10+ UML state machines widely used in the literature. The evaluation reveals that our tool can conduct model checking on those models efficiently. We also compared our tool with an existing tool HUGO. The results show that our tool can handle more features than HUGO and in all the models that can be processed by both tools, our tool outperforms HUGO.

This thesis provides methods and tools to help improve the quality of requirement specification and system designs. For requirement specifications, we noticed the communication barrier between stakeholders and formal specifications, therefore our work process natural language directly to improve the involvement of stakeholders. For system designs, we provide direct support of model checking on UML statemachines, which release users from learning any formal languages. Our methods are believed to be effective in quality improvement as confirmed with authors of the requirement specifications.

7.2 Future Work

In this section, we discuss the possible future directions that we are investigating.

We would like to improve the quality of artifacts, i.e., requirement specification and design models. Such artifacts, which are produced early in the software development process, are crucial to the success of the project.

In this thesis, we have explored techniques to improve the quality of both the requirement specification and existing design models. It is promising to look into how we can guide the process from requirement specifications to obtain design models and help to improve the quality of the initially created design models.

In Chapter 4, we obtained a DFA for an agent from the scenarios that describe the dynamic behaviors of the agent. This can serve as an initial guidance for creating the design models which capture dynamic behaviors, such as UML state machines. There are existing works [125, 132] which generate UML state machines from scenarios captured by MSC with the aid of manual annotation of invariants and states on MSC time lines. Inspired by their works, we can explore to annotate invariants on the DFA that is generated by our approach, which helps to group states with the same status and create hierarchy.

Another interesting direction is to extract invariants and properties from the use case descriptions , such as brief description sections, of use case documents. These invariants and properties are constraints that should be or must be preserved by the systems and can act as input to the model checking procedure. This further aids the improvement and validation process of system designs and can also be used together with approaches by Uchitel et al. [124] to synthesize behavior models.

Bibliography

[1] The Cadence SMV Model Checker. http://www.kenmcmil.com/smv.html.

[2] CRaG Systems Use Case Template. http://www.cragsystems.co.uk/development_process/docs/UseCaseDocumentTemplate.doc.

[3] The FDR Web Site. http://www.cs.ox.ac.uk/projects/concurrency-tools/.

[4] ITU-T Recommendation Z.120 Message Sequence Chart (MSC). http://www.itu.int/rec/T-REC-Z.120-201102-I/en.

[5] USMMC, A UML State Machines Model Checker, http://www.comp.nus.edu.sg/~lius87/.

[6] OMG Unified Language Superstructure Specification (formal). Version 1.4. http://www.omg.org/spec/UML/1.4/PDF/index.htm.

[7] OMG Unified Language Superstructure Specification (formal). Version 2.4.1. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/.

[8] OMG: XML Metadata Interchange (XMI v2.0). http://www.omg.org/spec/XMI/2.4.1/.

[9] Princeton University "About WordNet." WordNet. Princeton University. 2010. http://wordnet.princeton.edu.

[10] Proco Download Page. http://www.tcs.hut.fi/SMUML/.

[11] The Promela Manual Page. http://spinroot.com/spin/Man/promela.html.

[12] PVS Specification and Verification System. http://pvs.csl.sri.com/.

[13] The Spin Web Site. http://spinroot.com/spin/whatispin.html.

[14] Splitta (version 1.03), A Statistical Sentence Boundary Detection Tool. https://code.google.com/p/splitta/.

[15] Symbolic Methods for UML Behavioural Diagrams, http://www.tcs.hut.fi/Research/Logic/SMUML.shtml.

[16] UPPAAL Web Site. `http://www.uppaal.org/`.

[17] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification v1.1. `http://www.omg.org/spec/QVT/1.1/`, 2007.

[18] Hugo/RT Website. `http://www.pst.informatik.uni-muenchen.de/projekte/hugo/`, 2012.

[19] mCRL2, A Specification Language and Toolset. `http://www.mcrl2.org/release/user_manual/index.html`., 2012.

[20] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proceedings of 20th International Conference on Very Large DataBase*, pages 487–499, 1994.

[21] S. Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., 2002.

[22] É. André, C. Choppy, and K. Klai. Formalizing Non-Concurrent UML State Machines Using Colored Petri Nets. *ACM SIGSOFT Software Engineering Notes*, 37(4):1–8, 2012.

[23] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and computation*, 75(2):87–106, 1987.

[24] D. B. Aredo. Semantics of UML Statecharts in PVS. In *In Proceedings of the 12th Nordic Workshop on Programming Theory (NWPT'00)*, 2000.

[25] G. Bai, J. Hao, J. Wu, Y. Liu, Z. Liang, and A. Martin. TrustFound: Towards a Formal Foundation for Model Checking Trusted Computing Platforms. In *19th International Symposium on Formal Methods (FM)*, pages 110–126. 2014.

[26] L. Baresi and M. Pezzè. On Formalizing UML with High-level Petri Nets. *Concurrent Object-Oriented Programming and Petri Nets*, pages 276–304, 2001.

[27] M. E. Beato, M. Barrio-Solórzano, C. E. Cuesta, and P. de la Fuente. UML Automatic Verification Tool with Formal Methods. *Electronic Notes in Theoretical Computer Science*, 127(4):3–16, 2005.

[28] J. Beckers, D. Klünder, S. Kowalewski, and B. Schlich. Direct Support for Model Checking Abstract State Machines by Utilizing Simulation. In *Abstract State Machines, B and Z*, volume 5238 of *Lecture Notes in Computer Science*, pages 112–124. Springer Berlin Heidelberg, 2008.

[29] T. E. Bell and T. A. Thayer. Software Requirements: Are They Really a Problem? In *Proceedings of the 2Nd International Conference on Software Engineering*, pages 61–68. IEEE Computer Society Press, 1976.

[30] P. Bhaduri and S. Ramesh. Model Checking of Statechart Models: Survey and Research Directions. *arXiv preprint cs/0407038*, 2004.

[31] B. Boehm and V. R. Basili. Software Defect Reduction Top 10 List. *Computer*, 34(1):135–137, 2001.

[32] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *Computer*, 21(5):61–72, 1988.

[33] E. Börger, A. Cavarra, and E. Riccobene. Modeling the Dynamics of UML State Machines. In *Abstract State Machines-Theory and Applications*, pages 167–186. Springer, 2000.

[34] E. Börger, A. Cavarra, and E. Riccobene. Modeling the Meaning of Transitions from and to Concurrent States in UML State Machines. In *Proceedings of the ACM symposium on Applied computing*, SAC'03, pages 1086–1091, 2003.

[35] E. Börger, A. Cavarra, and E. Riccobene. On Formalizing UML State Machines Using ASMs. *Information Software Technology*, 46(5):287, 2004.

[36] C. Choppy, K. Klai, and H. Zidani. Formal Verification of UML State Diagrams: A Petri Net Based Approach. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.

[37] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV'02, pages 359–364. Springer-Verlag, 2002.

[38] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[39] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 2000.

[40] K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An Automatic Verification Tool for UML. Technical Report CSE-TR-423-00, University of Michigan, 2000.

[41] C. Damas, B. Lambeau, P. Dupont, and A. Van Lamsweerde. Generating Annotated Behavior Models from End-User Scenarios. *IEEE Transactions on Software Engineering*, 31(12):1056–1073, 2005.

[42] C. Damas, B. Lambeau, and A. Van Lamsweerde. Scenarios, Goals, and State Machines: a Win-Win Partnership for Model Synthesis. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 197–207. ACM, 2006.

[43] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In *Formal Methods for Components and Objects*, volume 2852 of *Lecture Notes in Computer Science*, pages 71–98. Springer Berlin Heidelberg, 2003.

[44] G. Del Castillo and K. Winter. Model Checking Support for the ASM High-level Language. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346, 2000.

[45] W. Dong, J. Wang, X. Qi, and Z.-C. Qi. Model Checking UML Statecharts. In *Eighth Asia-Pacific Software Engineering Conference, APSEC 2001.*, pages 363–370, 2001.

[46] J. Dubrovin. Jumbala — An Action Language for UML State Machines. Technical report, Helsinki University of Technology, Laboratory for Theoretical Computer Science, 2006.

[47] J. Dubrovin and T. Junttila. Symbolic Model Checking of Hierarchical UML State Machines. Technical Report B23, Helsinki University of Technology, 2007.

[48] B. Egon and S. Robert. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag USA., 2003.

[49] R. Eshuis and R. Wieringa. Requirements-Level Semantics for UML Statecharts. In *Formal Methods for Open Object-Based Distributed Systems IV*, volume 49 of *IFIP Advances in Information and Communication Technology*, pages 121–140. Springer US, 2000.

[50] A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Applications of Linguistic Techniques for Use Case Analysis. *Requirements Engineering*, 8(3):161–170, 2003.

[51] H. Fecher and J. Schönborn. UML 2.0 State Machines: Complete Formal Semantics via Core State Machine. *Formal Methods: Applications and Technology*, pages 244–260, 2007.

[52] H. Fecher, J. Schönborn, M. Kyas, and W. de Roever. 29 New Unclarities in the Semantics of UML 2.0 State Machines. *Formal Methods and Software Engineering*, pages 52–65, 2005.

[53] D. Firesmith. Specifying Good Requirements. *Journal of Object Technology*, 2:77–87, 2003.

[54] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., 3 edition, 2003.

[55] V. Gervasi. The Cico Domain-Based Parser. Technical report, Technical Report TR-01-25, University of Pisa, Dipartimento di Informatica, November, 2001.

[56] V. Gervasi and D. Zowghi. Reasoning about Inconsistencies in Natural Language Requirements. *ACM Transactions on Software Engineering Methodololy*, 14(3):277–330, 2005.

[57] S. Gnesi, D. Latella, and M. Massink. Model Checking UML Statechart Diagrams Using JACK. In *Proceedings 4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 46–55. IEEE, 1999.

[58] L. Gui, J. Sun, Y. Liu, Y. J. Si, J. S. Dong, and X. Y. Wang. Combining model checking and testing with an application to reliability prediction and distribution. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 101–111. ACM, 2013.

[59] J. Gutiérrez, C. Nebut, M. Escalona, M. Mejías, and I. Ramos. Visualization of Use Cases Through Automatically Generated Activity Diagrams. *Model Driven Engineering Languages and Systems*, pages 83–96, 2008.

[60] H. H. Hansen, J. Ketema, B. Luttik, M. Mousavi, and J. P. van de. Towards Model Checking Executable UML Specifications in mCRL2. *Innovations in Systems and Software Engineering*, 6(1-2):83–90, 2010. Open Access.

[61] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of computer programming*, 8(3):231–274, 1987.

[62] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30:31–42, 1997.

[63] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[64] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[65] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[66] H. F. Hofmann and F. Lehner. Requirements Engineering As a Success Factor in Software Projects. *IEEE Software*, 18(4):58–66, 2001.

[67] I. Jacobson. Use cases – Yesterday, Today, and Tomorrow. *Software and Systems Modeling*, 3(3):210–220, 2004.

[68] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering–A Use Case Driven Approach.* Addison-Wesley, 1992.

[69] I. Jacobson, I. Spence, and K. Bittner. Use Case 2.0 The Guide to Succeeding with Use Cases. `http://www.ivarjacobson.com/Use_Case2.0_ebook/`, December 2011.

[70] J. Janneck and P. Kutter. *Mapping automata: simple abstract state machines.* TIK-Report. Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zürich (ETH), 1998.

[71] K. Jensen and L. Kristensen. *Coloured Petri Nets: Modeling and Validation of Concurrent Systems.* Springer-Verlag New York Inc, 2009.

[72] Y. Jin, R. Esser, and J. Janneck. A Method for Describing the Syntax and Semantics of UML Statecharts. *Software and Systems Modeling*, 3(2):150–163, 2004.

[73] J. Jürjens. A UML Statecharts Semantics with Message-Passing. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 1009–1013. ACM.

[74] T. Jussila, J. Dubrovin, T. Junttila, T. Latvala, and I. Porres. Model Checking Dynamic and Hierarchical UML State Machines. *Proc. MoDeV2a: Model Development, Validation and Verification*, pages 94–110, 2006.

[75] A. Knapp and S. Merz. Model Checking and Code Generation for UML State Machines and Collaborations. In *Proceedings of 5th Workshop on Tools for System Design and Verification*, volume 11, pages 59–64, 2002.

[76] A. Knapp, S. Merz, and C. Rauh. Model Checking - Timed UML State Machines and Collaborations. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'02)*, pages 395–416. Springer-Verlag, 2002.

[77] L. Kof. Scenarios: Identifying Missing Objects and Actions by Means of Computational Linguistics. In *15th IEEE International Requirements Engineering Conference. RE'07.*, pages 121–130. IEEE, 2007.

[78] L. Kof. Translation of Textual Specifications to Automata by Means of Discourse Context Modeling. *Requirements Engineering: Foundation for Software Quality*, pages 197–211, 2009.

[79] J. Kong, K. Zhang, J. Dong, and D. Xu. Specifying Behavioral Semantics of UML Diagrams Through Graph Transformations. *Journal of Systems and Software*, 82(2):292–306, 2009.

[80] G. Kwon. Rewrite Rules and Operational Semantics for Model Checking UML Statecharts. In *Proceedings of the 3rd International Conference on the Unified Modeling Language: advancing the standard (UML'00)*, pages 528–540. Springer-Verlag, 2000.

[81] V. S. Lam and J. Padget. Symbolic Model Checking of UML Statechart Diagrams with an Integrated Approach. In *Proceedings. 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems.*, 2004.

[82] V. S. Lam and J. Padget. An Integrated Environment for Communicating UML Statechart Diagrams. In *Proceedings of the ACS/IEEE 2005 International Conference on Computer Systems and Applications*, AICCSA '05, pages 111–vii. IEEE Computer Society, 2005.

[83] K. G. Larsen and B. Thomsen. A Modal Process Logic. In *Proceedings of the Third Annual Symposium on Logic in Computer Science, LICS '88.*, pages 203 – 210, 1988.

[84] D. Latella, I. Majzik, and M. Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model Checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[85] D. Latella, I. Majzik, and M. Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 1, page 465, 1999.

[86] C. M. Y. Lawrence Bernstein. *Trustworthy Systems Through Quantitative Software Engineering*. Wiley-IEEE Computer Society Press, October 2005.

[87] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements Specification for Process-Control Systems. *IEEE Transanctions on Software Engineering*, 20(9):684–707, 1994.

[88] J. Lilius and I. P. Paltor. The Semantics of UML State Machines. Technical report, Turku Centre for Computer Science, 1999.

[89] J. Lilius and I. P. Paltor. vUML: A Tool for Verifying UML Models. *14th IEEE International Conference on Automated Software Engineering*, pages 255–258, 1999.

[90] S. Liu. Automatic Specification-based Testing: Challenges and Possibilities. In *Fifth International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 5–8. IEEE, 2011.

[91] S. Liu, Y. Liu, É. André, C. Choppy, J. Sun, B. Wadhwa, and J. S. Dong. A Formal Semantics for Complete UML State Machines with Communications. In *Integrated Formal Methods (iFM)*, pages 331–346, 2013.

[92] S. Liu, Y. Liu, J. Sun, M. Zheng, B. Wadhwa, and J. S. Dong. USMMC: A Self-contained Model Checker for UML State Machines. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 623–626. ACM, 2013.

[93] S. Liu, J. Sun, Y. Liu, Y. Zhang, B. Wadhwa, J. S. Dong, and X. Wang. Automatic Early Defects Detection in Use Case Documents. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 785–790. ACM, 2014.

[94] Y. Liu, X. Zhang, J. S. Dong, Y. Liu, J. Sun, J. Biswas, and M. Mokhtari. Formal Analysis of Pervasive Computing Systems. In *Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on*, pages 169 –178, july 2012.

[95] Y. Liu, X. Zhang, Y. Liu, J. Dong, J. Sun, J. Biswas, and M. Mokhtari. Towards Formal Modelling and Verification of Pervasive Computing Systems. In R. Kowalczyk and N. T. Nguyen, editors, *Transactions on Computational Collective Intelligence XVI*, Lecture Notes in Computer Science, pages 62–91. Springer Berlin Heidelberg, 2014.

[96] B. Lundell, B. Lings, A. Persson, and A. Mattsson. UML Model Interchange in Heterogeneous Tool Environments: An Analysis of Adoptions of XMI 2. In *MODELS*, pages 619–630. 2006.

[97] E. Mäkinen and T. Systä. Minimally Adequate Teacher Synthesizes Statechart Diagrams. *Acta Informatica*, 38(4):235–259, 2002.

[98] M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[99] C. Mats and J. Lars. Formal Verification of UML-RT Capsules using Model Checking. Master's thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Sweden, 2009.

[100] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University Pittsburgh, 1992. UMI Order No. GAX92-24209.

[101] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[102] C. Nebut, F. Fleurey, Y. L. Traon, and J. M. Jézéquel. A Requirement-Based Approach to Test Product Families. In *Proc. Fifth Workshop Product Families Eng*, pages 198–210. Springer Verlag, 2003.

[103] M. Y. Ng and M. Butler. Tool Support for Visualizing CSP in UML. In *Formal Methods and Software Engineering*, volume 2495 of *Lecture Notes in Computer Science*, pages 287–298. Springer Berlin Heidelberg, 2002.

[104] M. Y. Ng and M. Butler. Towards Formalizing UML State Diagrams in CSP. *Third IEEE International Conference on Software Engineering and Formal Methods, SEFM'03*, 0:138, 2003.

[105] R. Nicola and F. Vaandrager. Action Versus State Based Logics for Transition Systems. In *Semantics of Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Berlin Heidelberg, 1990.

[106] J. Nivre. Algorithms for Deterministic Incremental Dependency Parsing. *Computational Linguistics*, 34(4):513–553, 2008.

[107] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating Test Data from State-based Specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, 2003.

[108] J. Oncina and P. Garcia. Identifying Regular Languages In Polynomial Time. In *Advances in Structural and Syntactic Pattern Recognition*, volume 5 of *Machine Perception and Artificial Intelligence*, pages 99–108. World Scientific, 1992.

[109] R. G. Pettit, IV, and H. Gomaa. Validation of Dynamic Behavior in UML Using Colored Petri Nets. In *Proceedings of UML' 2000 workshop, Dynamic behavior in UML models: semantic quesions*, volume 1939, pages 295–302. Springer Verlag, 2000.

[110] M. O. Rabin and D. Scott. Finite Automata and Their Decision Problems. *IBM journal of research and development*, 3(2):114–125, 1959.

[111] A. Raji and P. Dhaussy. User Context Models - A Framework to Ease Software Formal Verifications. In *ICEIS (3)*, pages 380–383. SciTePress, 2010.

[112] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines - A Lightweight Formal Approach. In *Proc. FASE 2000, number 1783 in Lecture Notes in Computer Science*, pages 127–146. Springer Verlag, 2000.

[113] R. L. Rivest and R. E. Schapire. Inference of Finite Automata Using Homing Sequences. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, pages 411–420. ACM, 1989.

[114] T. Schäfer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):357–369, 2001.

[115] J. Schönborn. Formal Semantics of UML 2.0 Behavioral State Machines. Technical report, Institute of Computer Science and Applied Mathematics, Christian-Albrechts-University of Kiel, 2005.

[116] W. Shen, K. Compton, and J. Huggins. A Toolset for Supporting UML Static and Dynamic Model Checking. In *Proceedings. 26th Annual International Computer Software and Applications Conference. (COMPSAC 2002)*, pages 147–152, 2002.

[117] A. Sinha, A. Paradkar, P. Kumanan, and B. Boguraev. A Linguistic Analysis Engine for Natural Language Use Case Description and its Application to Dependability Analysis in Industrial Use Cases. In *IEEE/IFIP International Conference on Dependable Systems & Networks, 2009. DSN '09.*, pages 327 – 336, 2009.

[118] A. Sinha, S. M. Sutton, and A. Paradkar. Text2Test: Automated Inspection of Natural Language Use Cases. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 155–164. IEEE Computer Society, 2010.

[119] M. Spielmann. Model Checking Abstract State Machines and Beyond. In *Abstract State Machines - Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 323–340. Springer Berlin Heidelberg, 2000.

[120] J. Sun, Y. Liu, J. Dong, and J. Pang. PAT: Towards Flexible Verification Under Fairness. In *Computer Aided Verification*, pages 709–714. Springer, 2009.

[121] F. Törner, M. Ivarsson, F. Pettersson, and P. Öhman. Defects in Automotive Use Cases. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 115–123. ACM, 2006.

[122] I. Traoré. An Outline of PVS Semantics for UML Statecharts. *Journal of Universal Computer Science*, 6:2000, 2000.

[123] J. Trowitzsch and A. Zimmermann. Real-Time UML State Machines: An Analysis Approach. *Object oriented software design for real time and embedded computer systems*, 2005.

[124] S. Uchitel, G. Brunet, and M. Chechik. Synthesis of Partial Behavior Models from Properties and Scenarios. *IEEE Transactions on Software Engineering*, 35(3):384–406, 2009.

[125] S. Uchitel, J. Kramer, and J. Magee. Synthesis of Behavioral Models from Scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, 2003.

[126] S. Uchitel, J. Kramer, and J. Magee. Incremental Elaboration of Scenario-based Specifications and Behavior Models Using Implied Scenarios. *ACM Transactions on Software Engineering Methodololy*, 13(1):37–85, 2004.

[127] K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. Prod Reference Manual. Technical report, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, 1995.

[128] M. Von Der Beeck. A Structured Operational Semantics for UML Statecharts. *Software and Systems Modeling*, 1(2):130–141, 2002.

[129] T. Wang, S. Song, J. Sun, Y. Liu, J. S. Dong, X. Wang, and S. Li. More Anti-chain Based Refinement Checking. In *ICFEM*, 2012.

[130] J. Whittle. Specifying Precise Use Cases with Use Case Charts. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 290–301. Springer Berlin Heidelberg, 2006.

[131] J. Whittle and P. Jayaraman. Generating Hierarchical State Machines from Use Case Charts. In *Proceedings of the 14th IEEE International Requirements Engineering Conference*, number 16-25. IEEE Computer Society, 2006.

[132] J. Whittle and J. Schumann. Generating Statechart Designs from Scenarios. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 314–323. ACM, 2000.

[133] M. Wulf De, L. Doyen, T. Henzinger, and J.-F. Raskin. Antichains: A New Algorithm for Checking Universality of Finite Automata. In *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 17–30. Springer Berlin Heidelberg, 2006.

[134] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie. Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 12:1–12:11. ACM, 2012.

[135] T. Yue, S. Ali, and L. Briand. Automated Transition from Use Cases to UML State Machines to Support State-based Testing. In *Proceedings of the 7th European conference on Modelling foundations and applications*, ECMFA'11, pages 115–131. Springer-Verlag, 2011.

[136] T. Yue, L. Briand, and Y. Labiche. A Use Case Modeling Approach to Facilitate the Transition towards Analysis Models: Concepts and Empirical Evaluation. In *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 484–498. Springer Berlin Heidelberg, 2009.

[137] T. Yue, L. Briand, and Y. Labiche. An Automated Approach to Transform Use Cases into Activity Diagrams. *Modelling Foundations and Applications*, pages 337–353, 2010.

[138] A. Zeichick. Modeling Usage Low; Developers Confused About UML 2.0, MDA. Technical report, BZ Research, 2002.

[139] S. Zhang and Y. Liu. An Automatic Approach to Model Checking UML State Machines. In *Fourth International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI-C)*, pages 1–6. IEEE, 2010.

[140] Y. Zhang and S. Clark. Syntactic Processing Using the Generalized Perceptron and Beam Search. *Computational Linguistics*, 37(1):105–151, 2011.

Appendix A

Auxiliary Definitions on UML State Machine Semantics

We provide the auxiliary functions and definitions that are used in Chapter 5 in this appendix.

Since UML state machine has a hierarchical structure, we define functions $isAncestor$ which decide the ancestor/decedent relations between states and regions.

$isAncestor : S \cup R \times S \cup R \cup PS \cup S_f \rightarrow \mathbb{B}$ is defined as follows:

$$isAncestor(s, s') \triangleq \begin{cases} True, & if\ (s \in S \wedge \exists\, r \in s.\widehat{r} : isAncestor(r, s')) \vee \\ & (s \in R \wedge \exists\, s_0 \in s.\widehat{v} : isAncestor(s_0, s')) \vee s == s' \\ False, & otherwise \end{cases}$$

where $isAncestor(s, s') == T$ represents that s is the ancestor of s'. The isAncestor relation is transitive.

Definition 32 (Sequence Transition) *A sequence transition(st) is an ordered list of transitions. It is defined as: (1)*$\forall\, t \in T, t \in ST$*; (2)*$\forall\, st_i, st_j \in ST : last(st_i) = first(st_j) \Rightarrow$

$st_i \frown st_j \in ST$.

A sequence transition is an ordered list of transitions such that the sibling transitions are linked head-to-tail. The compound transition is a special case of sequence transition where the sources of the first transition and the targets of the last transition are constrained to be states.

Least Common Ancestor (LCA) is an operation defined for a state machine. It returns the smallest common ancestor of the given set of vertices.

Definition 33 (Least Common Ancestor) $LCA : \mathbb{P}\, V \to R \cup S_o \cup PS$ *Formally,*

$$LCA(\widehat{V}) \triangleq \begin{cases} s, & \textbf{\textit{if}}\ \exists\, s \in \widehat{V} : (s \in S_o \wedge (\forall\, s' \in \widehat{V}, isAncestor(s, s'))) \\ r, & \textbf{\textit{if}}\ \nexists s \in \widehat{V} : s \in S_o \wedge (\forall\, s' \in \widehat{V}, isAncestor(s, s')) \wedge \\ & \exists\, r \in R \wedge (\forall\, s \in \widehat{V}, isAncestor(r, s)) \wedge \\ & (\forall\, r' \in R : \forall\, s \in \widehat{V}, isAncestor(r', s)) \Rightarrow isAncestor(r', r) \end{cases}$$

LCA of a set of vertices can be a region or an orthogonal composite state. We need to guarantee "least" in both situations. If a region is the LCA of a set of states, we guarantee "least" by constraint $\forall\, r' \in R : \forall\, s \in \widehat{V}, isAncestor(r', s)) \Rightarrow isAncestor(r', r)$. This constraint indicates that if both region r and r' are ancestors of the set of vertices $\widehat{V}$, then r' must also be the ancestor of r, which implies that r is the "least" among all the ancestors of $\widehat{V}$.

Main Source/Target The Main source of a transition is the outermost vertex exited by the transition on firing and is defined by function $MainSource : T \to V$

Definition 34 (Main Source) *Let* $S(t) = t.\widehat{sv} \cup t.\widehat{tv}$
$MainSource(t) \triangleq$

$$\begin{cases} LCA(S(t)), & \textbf{\textit{if}}\ LCA(S(t)) \notin R \\ s, & \textbf{\textit{if}}\ LCA(S(t)) \in R \wedge ((!isJoin(t) \wedge \exists\, s \in LCA(S(t)).\widehat{v} : isAncestor(s, t.sv)) \vee \\ & (isJoin(t) \wedge \exists\, s \in LCA(S(t)).\widehat{v} : (\forall\, s' \in t.\widehat{sv} \backslash \{t.sv\} \Rightarrow isAncestor(s, s')))) \end{cases}$$

The Main target of a transition is the outermost vertex entered by a transition on firing and is defined by function $MainTarget : T \rightarrow V$

Definition 35 (Main Target)

$MainTarget(t) \triangleq$

$$\begin{cases} LCA(S(t)), & \textbf{\textit{if}}\ LCA(S(t)) \notin R \\ s, & \textbf{\textit{if}}\ LCA(S(t)) \in R \wedge ((!isFork(t) \wedge s \in LCA(S(t)).\widehat{v} \wedge isAncestor(s, t.tv)) \vee \\ & (isFork(t) \wedge s \in LCA(S(t)).\widehat{v} \wedge (\forall\, s' \in t.\widehat{tv} \backslash \{t.tv\} \Rightarrow isAncestor(s, s')))) \end{cases}$$

Main source and main target are defined to capture the ordering of the set of states exited and entered when firing a transition. If the LCA of the source and target states of a transition is an orthogonal composite state, then the main source/target is the orthogonal state. Otherwise, it is the direct subvertex of the LCA that contains the all the source states of t.

Enable: $\mathcal{K} \times E \rightarrow \mathbb{P}\, ST$ is a function which evaluates the enabled compound transitions under the current configuration and triggering event.

Definition 36 (Enable)

$$Enable(k, e) \triangleq \{st | st \in \widetilde{T} \vee (st \in ST \wedge st.tv \in C_{ps} \wedge st.\widehat{sv} \backslash sv \subset S \cup C_{ps}) \wedge st.\widehat{sv} \subset k \wedge (\forall\, i \in [1, len(st)] : enable(seg(st, i), e, k.V))\}$$

$$enable(t, e, GV) \triangleq \begin{cases} True, & \textbf{\textit{if}}\ Evaluate(t.g, GV) == T \wedge (e \in t.\widehat{tg} \vee t.\widehat{tg} = \varnothing) \\ False, & otherwise \end{cases}$$

$Enable(k, e)$ returns the set of sequence transitions of which all the guards of all its segment transitions are evaluated to true under the current configuration k, and the dispatched event

matches its triggering event e. Choice pseudostate requires explicit information to continue the current RTC step and is considered as a temporal stop. When a compound transition with n choice vertices is encountered, $n+1$ times of *Enable* function is called. The function $enable(t, e, V)$ determines whether the transition t can be triggered by event e under the current shared variable values GV.

Definition 37 (Leave) *$\mathcal{K}_{\mathcal{V}} \times T \to \mathbb{P}\, S \cup S_f \cup PS$ maps a transition to the set of vertices it leaves on firing. Formally:*

$$Leave(kv, t) \triangleq \begin{cases} \varnothing, & \textbf{\textit{if}}\ isAncestor(t.sv, t.\iota) \\ Lv(MainSource(t), t.\widehat{sv}, kv), & otherwise \end{cases}$$

$$Lv(s, \widehat{s}, kv) \triangleq \begin{cases} \{s''\} \cup Lv(s'', \widehat{s}, kv), & \textbf{if}\ s \in R \wedge \{s''\} = s.\widehat{v} \cap kv \\ \{s\} \cup \bigcup_{r \in s.regions} Lv(r, \widehat{s}, kv), & \textbf{if}\ s \in S_c \vee s \in S_o \\ s' & \textbf{if}\ s \in CR \wedge \exists\, ex \in s.\widehat{ex} \wedge \exists\, s' \in S : ex.\iota \in s'.\widehat{r} \\ \{s\}, & otherwise \end{cases}$$

For internal (transitions which do not leave or enter any states) and local transitions (both satisfies the constrain $isAncestor(s, t.\iota)$), the set of vertices they left is empty set (indicated by $\varnothing$). For external transitions, starting from the main source state, function Lv recursively compute the set of vertices left on firing the transition. If a region is exited by a fired compound transition, then all the current active vertices within the region are exited in an innermost-out order. If a composite state is exited, all its (orthogonal) regions are exited in an innermost-out order. If a connection point reference is exited, the submachine state it is defined is exited, which means the state machine or composite state represented by the submachine state is exited. For simple states, final states and all the other kinds of pseudostates, only the vertex itself is exited.

Definition 38 (Enter) *$T \to \mathbb{P}(S \cup S_f \cup PS)$ maps a transition to the set of vertices it*

enters on firing. Formally,

$$Enter(t) \triangleq \begin{cases} \varnothing, & \textbf{\textit{if}}\ isAncestor(t.tv, t.\iota) \\ enter(MainTarget(t), t.\widehat{tv}), & otherwise \end{cases}$$

where $enter : (S \cup S_f \cup PS \cup RCR) \times \mathbb{P}(S \cup S_f \cup PS) \rightarrow \mathbb{P}(S \cup S_f \cup PS)$ *returns all the transitively activated states due to execution of a transition. Formally,*

$$enter(ms, \widehat{s}) \triangleq$$
$$\begin{cases} \{ms\} \cup \bigcup_{r \in s.\widehat{r}} enter(r, \widehat{s}), & \textbf{\textit{if}}\ ms \in S_c \vee ms \in S_o \\ enter(s', \widehat{s}), & \textbf{\textit{if}}\ ms \in R \wedge (\exists\, s \in \widehat{s} : isAncestor(ms, s)) \wedge \\ & \exists\, s' \in ms.\widehat{v} \wedge isAncestor(s', s) \\ enter(initial(ms), \widehat{s}), & \textbf{\textit{if}}\ ms \in R \wedge (\forall\, s \in \widehat{s}, !isAncestor(ms, s)) \\ defhistory(ms), & \textbf{\textit{if}}\ (ms \in SH_{ps} \vee ms \in DH_{ps}) \wedge \\ & (ms.\widehat{h} = \varnothing \vee \exists\, s \in ms.\widehat{h} : s \in S_f) \\ ms.\widehat{h} & \textbf{\textit{if}}\ (ms \in SH_{ps} \vee ms \in DH_{ps}) \wedge \\ & ms.\widehat{h}! = \varnothing \wedge \nexists s \in ms.\widehat{h} : s \in S_f) \\ ms.s & \textbf{\textit{if}}\ ms \in CR \wedge \exists\, en \in ms.\widehat{en} \wedge \exists\, s' \in S : en.\iota \in s'.\widehat{r} \\ \{ms\}, & otherwise \end{cases}$$

- For orthogonal (composite) states, the state is entered followed by all its containing regions.
- If the region contains one of the target states[1] of the transition, then the substate of the region which is the containers of the target state of the transition is entered.
- If the region does not contain any target states of the transition, then the state indicated by its initial pseudostate is entered.
- If a history state is encountered, and it is the first time for its containing state to be activated, or the last accessed state is a final state, the default history state will be entered.

[1] For a fork transition, there are multiple target states.

- If a history state is encountered and it is not the first time for its containing state to be activated, and the deepest contents[2] recorded by the history pseudostate are not all final states, then the recorded states will be entered.
- If a connection point reference is encountered, then the submachine state in which it is defined is entered, which indicates that the composite state/state machine referred to by the submachine state is entered.
- For simple states, final states and all the other kinds of pseudostates, only the vertex itself is entered.

Definition 39 (Conflict) $\mathcal{K}_{\mathcal{S}} \times \widetilde{T} \times \widetilde{T} \to \mathbb{B}$ *is a function which decides whether two compound transitions conflict with each other.*

$$
\begin{array}{l}
Conflict(ks, \tilde{t}, \tilde{t}') \triangleq \\
\left\{
\begin{array}{ll}
True, & (\bigcup_{i \in [1, len(\tilde{t})]} Leave(ks, seg(\tilde{t}, i)) = \varnothing \vee \\
 & (\bigcup_{i \in [1, len(\tilde{t})]} Leave(ks, (seg(\tilde{t}', i)) = \varnothing) \wedge \tilde{t}.\widehat{sv} \cap \tilde{t}'.\widehat{sv} \neq \varnothing) \vee \\
 & (\bigcup_{i \in [1, len(\tilde{t})]} Leave(ks, seg(\tilde{t}, i)) \cap (\bigcup_{i \in [1, len(\tilde{t})]} Leave(ks, (seg(\tilde{t}', i)) \neq \varnothing) \\
False, & otherwise
\end{array}
\right.
\end{array}
$$

There are two situations for two compound transition to be defined as conflicting:

> "Two transitions are said to conflict if they both exit the same state, or, more precisely, that the intersection of the set of states they exit is non-empty."
>
> [7, Chapter 15.3.12, Semantics, Conflicting transitions, p.575]

This is captured by

$(\bigcup_{i \in [1, len(\tilde{t})]} Leave(ks, seg(\tilde{t}, i)) \cap (\bigcup_{i \in [1, len(\tilde{t})]} Leave(ks, (seg(\tilde{t}', i)) \neq \varnothing.$

> "An internal transition in a state conflicts only with transitions that cause an exit from that state."
>
> [7, Chapter 15.3.12, Semantics, Conflicting transitions, p.575]

[2] The innermost states that is in the state hierarchy of the recorded states.

This is captured by

$(\bigcup_{i\in[1,len(\tilde{t})]} Leave(ks, seg(\tilde{t}, i)) = \varnothing \vee (\bigcup_{i\in[1,len(\tilde{t})]} Leave(ks, (seg(\tilde{t}', i)) = \varnothing) \wedge \tilde{t}.\widehat{sv} \cap \tilde{t}'.\widehat{sv} \neq \varnothing)$.

$\widetilde{T} \times \widetilde{T} \times \mathcal{K}_{\mathcal{S}}$ is a partial relation between two compound transitions. A pair of transitions $(\tilde{t}, \tilde{t}') \in Priority$ in the current active state configuration ks means that the first segment of transition $\tilde{t}$ has a smaller distance to the innermost simple state in the current active state configuration than the first segment of transition $\tilde{t}'$.

Definition 40 (Priority)

$$Priority \triangleq \{(\tilde{t}, \tilde{t}', ks) \mid \exists\, s \in \tilde{t}.\widehat{sv} : \forall\, s' \in \tilde{t}'.\widehat{sv}, \\ distance(s, Innermost(ks)) < distance(s', Innermost(ks))\}$$

where $Innermost(ks) \triangleq \{s \mid s \in S_s \wedge s \in ks\}$, $distance(s, \widehat{s}) \triangleq n$,
if $\forall\, s' \in \widehat{s}, max \mid s, s' \mid = n$.

Operation $\mid s, s' \mid$ represents the levels of regions between the two states s, s' in the state hierarchy. If both s and s' are simple states, the value of $\mid s, s' \mid$ is 0 regardless of whether s and s' are the same states or not. In this case, the priority cannot be decided. This is a semantic variation point in UML state machine v2.4.1 specification. In our approach, we consider such situation as a non-determinism and enumerate all possibilities. The *distance* operator returns the maximum distance between a composite state s and all simple states in the set $\widehat{s}$, which are transitively contained in the composite state s. Since states from orthogonal regions are not comparable, this function guarantees that the distance computation is consist with the algorithm described in [7, Chapter 15.3.12, Semantics, Firing Priorities, p.576]. The Priority of two compound transitions are decided by the priority of their first segment transitions.

Deferral conflict is the conflict about whether an event should be consumed or not. It is between a state which has deferral event defined and a source state of a transition which consumes the event.

Definition 41 (deferral Conflict) *Formally, function deferralConflict:* $T \times \mathcal{K} \times E$ *is defined as*

$$deferralConflict(t, k, e) \triangleq$$
$$\begin{cases} True, & \exists\, s, s' \in k.ks : (e \in s.\widehat{t_{def}} \wedge s' \in t.\widehat{sv} \backslash t.sv \wedge isAncestor(s', s)) \\ False, & otherwise \end{cases}$$

In our definition, we try to solve the conflict following the UML state machine specifications of [7], which is described as follows:

> "In case of a composite orthogonal state, substates of orthogonal regions may also introduce deferral conflicts. The conflict resolution follows the triggering priorities, where nested sates override enclosing states. In case of a conflict between states in different orthogonal regions, a consumer state overrides a deferring state"
>
> [7, Chaperer 15.3.11, Semantics, Deferred events]

Therefore in our definition of deferral conflict, we consider two situations.

- If the confliction does not involve orthogonal composite state, which means that the involved states must be in the same branch of state hierarchy, then we give higher priority to substates. This is captured by the first condition.
- If the confliction involves orthogonal composite state, we directly give higher priority to transitions which consumes the current event.

Definition 42 (Firable Transitions) *The function FirTrans :*$\mathcal{K} \times Trig \rightarrow \mathbb{P}\, \widetilde{T}$ *returns the set of enabled transitions of which conflicts are solved by priority rules. Formally, we define*

Firable Transitions $FirTrans(k,e) \triangleq \{\tilde{t} \mid \exists\, \tilde{t} \in Enable(k,e) \wedge !deferralConflict(first(\tilde{t}),k,e) \wedge (\nexists \tilde{t}' \in FirTrans(k,e) : Conflict(k.ks,\tilde{t},\tilde{t}')) \wedge (\nexists \tilde{t}" \in Enable(k,e) \backslash FirTrans(k,e) : Conflict(k.ks,\tilde{t},\tilde{t}") \wedge Priority(\tilde{t}",\tilde{t},k.ks))\}$

The purpose of the function Firable Transition is to select the largest non-conflicting subset from Enabled transitions such that transitions in the selected subset are non-conflicting and have higher priorities over the conflicting ones in the remaining part. The first step of deciding firable transitions is to check the deferral conflict. $!deferralConflict(first(\tilde{t}),k,e)$ means there is no such confliction or the source state of the (compound) transition is assigned higher priority over the states which have the events deferred. This is the basic condition before we proceed to check conflicts between enabled transitions. The function is defined in a incremental way, i.e., selecting a subset from the set of Enabled transitions gradually, from the highest priority to the lowest priority. Condition $\nexists \tilde{t}' \in FirTrans(k,e) : Conflict(k.ks,\tilde{t},\tilde{t}'))$guarantees that the newly selected compound transition does not conflict with existing compound transitions in the Firable transition set. Condition $\nexists \tilde{t}" \in Enable(k,e) \backslash FirTrans(k,e) : Conflict(k.ks,\tilde{t},\tilde{t}") \wedge Priority(\tilde{t}",\tilde{t},k.ks)$ guarantees that each time a transition with the highest priority is selected.

Appendix B

Comaprison of Work on Model Checking UML State Machines

We provide a comparison on the supported features[1] of the translation-based approaches (Section 6.3.1) in Table B.2. The symbol "$\surd$" denotes that the feature is supported, "$\times$" means the feature is not supported, "$\circ$" means the featured is discussed in the paper, however not all possible situations are considered. For example, for "conflict/priority", some works considered conflict among enabled transitions, but did not discuss conflicts due to deferred events. In this case, we regard the features to be partially supported.

We can conclude from the table that in the translation based approaches, time, submachine state, entry/exit pseudostate and junction pseudostate are the least commonly supported features. Less than 5 out of all the surveyed approaches support these features. No approach, which uses ASM as target language supports, choice or time features. But they almost all support orthogonal composite state, completion event and entry/exit behaviors. All

[1]For space consideration, we remove the features that are commonly supported by all approaches, such as simple state, transitions, initial pseudostate, etc.

Work	Target Language	Tool developed	UML version
[33]	Abstract state machines	×	1.3
[34]	Abstract state machines	×	1.3
[73]	Abstract state machines	×	1.4
[40]	Abstract state machines	√	<= 1.3
[72]	GDTL	√ (Moses)	1.5
[109]	Colored Petri net	√	<= 1.3
[26]	High-level Petri net	×	–
[36]	Colored Petri net	√(CPN-AMI)	–
[22]	Colored Petri net	×	2.2
[123]	Stochastic Petri net	×	2.0
[84]	PROMELA	×	1.1
[74]	PROMELA	√ (PROCO)	1.4
[114]	PROMELA	√(HUGO)	1.4
[99]	PROMELA	√(RSARTE)	–
[80]	SMV input language	×	1.3
[47]	NuSMV input language	√	–
[27]	SMV input language	√	< 1.3
[82]	NuSMV input language	√($SC2PiCal$)	1.5
[139]	CSP#	√	2.2
[76]	Timed automata	√ (HUGO/RT)	1.4
[60]	mCRL2	×	2.2
[104]	CSP	√	1.4
[24]	PVS	×	1.3
[122]	PVS	√($PrUDE$)	1.3

Table B.1: Summary of translation based approaches

approaches use Petri Net as target language do not support priority mechanism, defer and completion events. Seen from Table B.1, we can notice that only 5 surveyed translation approaches focus on UML2.x state machine specifications and only one tool is developed for model checking UML2.x state machines.

work	multiple charts	conflict (priority)	time	entry/exit behaviors	states		pseudostates					events		
					orthogonal	submachine	fork/join	junction	choice	history	entry/exit	defer	comp	call
[33]	×	√	×	√	∘	×	×	×	×	√	×	√	√	×
[34]	×	√	×	√	√	×	×	×	×	√	×	√	√	×
[73]	√	×	×	√	√	×	×	×	×	×	×	×	∘	√
[40]	×	×	×	×	√	×	×	×	×	×	×	×	√	×
[72]	×	√	×	√	√	×	√	√	×	√	×	√	√	×
[109]	√	×	×	×	×	×	×	×	×	×	×	×	×	√
[26]	√	×	×	×	×	×	×	×	×	×	×	×	×	√
[36]	×	×	×	×	√	×	×	×	×	×	×	×	×	×
[22]	×	×	×	√	×	×	×	×	∘	√	×	×	×	×
[123]	×	×	√	×	∘	×	√	√	√	×	×	×	×	×
[84]	×	∘	×	×	√	×	×	×	×	×	×	×	×	√
[74]	√	√	×	×	√	×	×	×	√	×	×	√	√	×
[114]	√	√	×	√	√	×	√	×	√	×	×	×	√	√
[99]	√	×	√	×	×	×	×	×	×	×	×	×	×	×
[80]	×	∘	×	×	√	×	×	×	×	×	×	×	×	×
[47]	√	×	×	×	√	×	×	×	√	×	×	√	√	×
[81]	√	√	×	×	×	×	×	×	×	×	×	×	×	×
[139]	×	×	×	√	√	√	√	×	×	√	√	×	√	×
[76]	×	×	×	√	×	×	×	×	√	×	×	×	√	×
[24]	×	×	×	√	√	√	√	√	√	√	√	×	×	√
[122]	×	√	√	×	√	×	√	√	√	√	×	×	√	√

Table B.2: UML state machines features supported by translation based approaches

work	syntax domain	action language	semantic domain	event queue	UML version
[85]	EHA	–	Kripke Structure	×	1.1
[128]	term	–	Kripke Structure	×	1.4
[45]	EHA	self-defined	Kripke Structure	×	1.1
[88]	term	–	–	×	1.3
[80]	term	abstract notation	Kripke Structure	×	1.3
[49]	set with functions	self-defined	LTS	×	1.3
[112]	–	–	LTS	√	<= 1.3
[43]	krtUML	self-defined	STS	√	1.4
[51]	core state machine	–	–	×	2.0
[115]	tuple	–	–	×	2.0
[47]	tuple	abstract notation	first-order logic	√	2.0

Table B.3: Syntax and Semantic domains of surveyed operational semantics

Table B.4 summarized the supported features of the approaches in Section 6.3.2. √ means the feature is supported, × means the feature is not supported, ∘ means the featured is discussed in the paper, however does not consider all the possible situations. ⊕ represents the situation that the corresponding feature is not directly formalized, but it is represented with other formally defined features in the semantics.

Table B.3 provides the syntax/semantic modals and action languages used by approaches which provide formal semantics for UML statemachines (Section 6.3.2).

work	multiple charts	conflict (priority)	time	entry/exit behaviors	states		pseudostates					events		
					orthogonal	submachine	fork/join	junction	choice	history	entry/exit	defer	comp	call
[85]	×	∘	×	×	√	×	×	×	×	×	×	×	×	√
[128]	×	∘	×	√	√	×	×	×	×	√	×	×	×	×
[45]	√	∘	×	√	√	×	×	×	×	×	×	×	√	–
[88]	×	√	√	√	√	×	×	×	×	√	×	√	√	√
[80]	×	∘	×	×	√	×	×	×	×	×	×	×	×	×
[49]	×	∘	×	√	√	×	×	×	×	×	×	×	√	√
[112]	√	∘	√	×	×	×	×	√	×	×	×	√	×	√
[43]	√	×	√	×	×	×	×	×	×	×	×	×	×	×
[51]	×	√	√	×	⊕	√	⊕	⊕	√	⊕	⊕	√	√	×
[115]	×	√	×	√	√	×	√	×	×	√	×	√	×	×
[47]	√	*no*	×	×	√	×	×	×	√	×	×	√	√	×

Table B.4: UML state machines features supported by semantic approaches